uc3m | Universidad Carlos III de Madrid

Grado Universitario en Ingeniería Informática
2017-2018

Trabajo Fin de Grado

Control de un robot usando Raspberry Pi y Planificación Automática

---

Alba Gragera Álvarez

Tutor:

Ángel García Olaya

Octubre 2018, Leganés

RESUMEN

La presente memoria corresponde al desarrollo del control de un robot utilizando Raspberry Pi y planificación automática mediante las herramientas ROS y PELEA. La primera es un *framework* para el desarrollo de software destinado a robots, que facilita el proceso proporcionando un conjunto de librerías y estructuras que se detallarán en el documento. La segunda, PELEA, es un software creado para la integración de planificación automática en robots que permite monitorizar la ejecución y conocer información acerca del estado del mundo.

A partir de esta idea se presentan en primer lugar la situación actual y el ámbito histórico de herramientas y sistemas como los robots, la Planificación Automática y las placas Raspberry Pi y Arduino, tecnologías que forman parte de la estructura del trabajo.

Dentro de esta revisión se identificarán también las debilidades y fortalezas presentes en estos campos y se realizarán comparaciones entre distintos métodos de implementación o herramientas con similar funcionamiento, justificando el porqué de la elección final. De esta forma se llegará a la definición de los objetivos y el alcance del proyecto, donde se fijarán las metas a alcanzar.

En el ámbito socio-económico se destaca el auge que está teniendo la robótica en nuestros tiempos y el gran desarrollo que aún se espera en este campo para el futuro, pese a que aún existen barreras a superar. Posteriormente y en función de los datos detallados se describe la solución adoptada para la implementación del sistema, mediante la especificación de los distintos casos de uso y los requisitos a tener en cuenta para su construcción. Una vez realizado, el funcionamiento del sistema será verificado por una serie de pruebas cuyos resultados serán estudiados, proponiendo mejoras adicionales en la sección de trabajos futuros.

Finalmente se comentan las conclusiones obtenidas tras la realización del proyecto, así como información relativa a la planificación del mismo y los costes que ha supuesto su desarrollo.

Palabras clave: Cliente; Servidor; Planificación Automática; ROS; PELEA.

AGRADECIMIENTOS

En primer lugar, agradecer a la Universidad Carlos III de Madrid y a todo su profesorado la formación recibida por su parte durante estos años, permitiéndome llegar hasta este punto.

Especialmente gracias a Ángel García Olaya, tutor de este Trabajo Fin de Grado, por ejercer de guía del mismo durante el desarrollo y por su continua disponibilidad y apoyo prestado.

Quiero dar las gracias también a mis padres por su preocupación y sus ánimos, a mi hermana por su paciencia, por cada rato que este trabajo me ha robado de jugar con ella, y a Ernesto, por sus consejos y por sufrirme todo este tiempo.

Y por su supuesto, gracias a todos los compañeros y amigos que han hecho que haya merecido la pena pasar por este camino.

ÍNDICE DE CONTENIDO

ÍNDICE DE FIGURAS

ÍNDICE DE TABLAS

1. INTRODUCCIÓN

Desde hace varios milenios, el ser humano se ha sentido atraído por la construcción de máquinas que, en cierta forma, pudiesen aliviar su carga de trabajo o ser útiles para cualquier otra actividad, surgiendo así por ejemplo los primeros relojes automatizados.

Pasando por diseños e ideas intermedias, estos conceptos han avanzado y se han desarrollado a lo largo del tiempo hasta lo que hoy se conoce como robótica. Especialmente durante los últimos años, este campo está teniendo un gran crecimiento debido a los desarrollos de las nuevas tecnologías y herramientas que lo componen. Así, se ha podido ver cómo la robótica ha pasado de ser utilizada para la realización de tareas repetitivas a comenzar a abarcar propósitos mucho más extensos.

De esta forma aparecen los primeros conceptos de Planificación Automática, creada para controlar el robot *Shakey*, el pionero en la tarea de guiarse de forma autónoma y sobre el que se hablará más adelante. Desde entonces la Inteligencia Artificial está presente en el día a día de las personas de muchas formas distintas, pero este documento se centrará en la robótica. Así, en estas páginas se podrá leer acerca de la "Cuarta Revolución Industrial", en la que se explica cómo está afectando (y afectará) la introducción de los robots en la sociedad, así como los desafíos que aún quedan por delante.

Lo cierto es que gracias a este tipo de sistemas inteligentes se han realizado grandes avances en investigaciones, como puede ser la misión *Mars Science Laboratory*, llevada a cabo por la NASA y en la que se situó el rover *Curiosity* sobre la superficie de Marte para la toma de fotografías y de muestras del suelo del planeta. Este tipo de vehículos serán comentados en la sección del estado del arte.

Observando este y otros ejemplos que se mencionarán más adelante, es evidente pensar que los campos de la robótica y la Inteligencia Artificial se encuentran en pleno crecimiento, uniéndose muchas veces en proyectos que dan lugar a agentes inteligentes.

Así surge este trabajo, cuyo objetivo es la creación de un sistema capaz de controlar un robot mediante Planificación Automática. En este documento se explica su proceso de desarrollo, comenzando con el contexto actual e histórico de las tecnologías utilizadas y el impacto socio-económico para terminar con los resultados y conclusiones obtenidas tras la finalización del proyecto.

1.1.Motivación

La motivación de este trabajo surge del auge de la robótica mencionado líneas más arriba y del interés que el grado en Ingeniería Informática ha despertado sobre el tema. Por ello se ha decidido implementar un sistema de control para un robot Pioneer 3 DX basado en Planificación Automática. Por cuestiones prácticas de comodidad y ahorro de espacio mencionadas más adelante, dicho software será implementado en una Raspberry Pi.

Con el desarrollo de este trabajo se pretende tener un primer contacto en el uso de herramientas útiles en el campo de la robótica, además de aprender sobre su forma de implementación. De esta forma se espera ampliar los conocimientos respecto a este campo de una forma teórica y práctica, para en un futuro tener la posibilidad de seguir trabajando sobre ello.

Igualmente, se estudiará el estado actual del campo para conocer las herramientas óptimas para su construcción y se distinguir las fortalezas y debilidades de cada una.

1.2.Estructura del documento

Para explicar todos los puntos relacionados con el desarrollo del trabajo, así como la información relativa al mismo, se ha dividido esta memoria en los siguientes bloques:

- Introducción: es la presente sección, donde se incluye una breve descripción acerca del trabajo.
- Estado del arte: se trata de la explicación acerca de la historia y estado actual de las tecnologías utilizadas en el trabajo.
- Objetivos y alcance: muestra las principales metas a alcanzar por el sistema desarrollado.
- Materiales y métodos: detalla las herramientas finalmente utilizadas y la justificación de su empleo.
- Entorno socio-económico: contiene información acerca del impacto social y económico producido por los avances de la robótica.
- Diseño e implementación: se trata de la descripción del sistema creado. Contiene la especificación de los casos de uso y los requisitos, así como información acerca del proceso de desarrollo.
- Resultados: se realiza una comparación de los resultados obtenidos con los esperados a través de una serie de pruebas realizadas.
- Conclusiones: apartado que incluye las impresiones finales tras la realización del trabajo. Contiene también las dificultades encontradas durante el mismo.

- Trabajos futuros: posibles mejoras o desarrollos futuros a implementar sobre el resultado final.
- Planificación y presupuesto: muestra las planificaciones realizadas para el proyecto, así como el coste del desarrollo.
- Bibliografía: contiene las referencias bibliográficas consultadas.
- Anexos: información adicional al trabajo.

2. ESTADO DEL ARTE

Antes de entrar en detalles sobre el trabajo es conveniente conocer la historia y el estado actual de la tecnología involucrada en el desarrollo, para comprender así sus fortalezas y debilidades.

2.1.Robótica

El término *robot* tuvo su primera aparición en el año 1921 en la obra *R.U.R (Rossum's Universal Robots)* [1], una representación teatral escrita por Karel Čapek. Dicho término proviene de la palabra checa *robota*, cuyo significado es 'trabajo forzado'. Tras el éxito de la obra esta expresión comenzó a usarse en todas las lenguas, reemplazando así al utilizado hasta entonces, autómata.

Pero hasta ese momento ya habían sido muchos los intentos de crear artilugios mecanizados, como *cresydra* [2], un reloj de agua diseñado en torno al año 230 a.C por el físico e inventor griego Ctesibus, cuyo funcionamiento se basa en una serie de piezas móviles capaces de medir el tiempo gracias al agua que fluye a velocidad constante a través de una abertura.

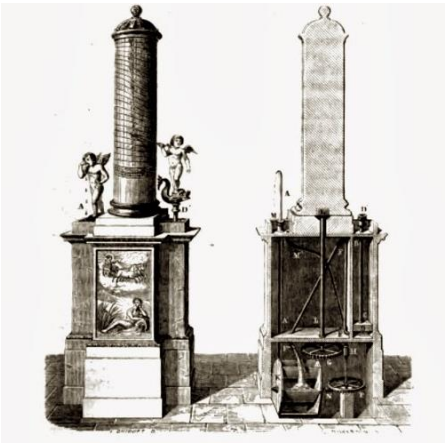


Fig. 2.1. Cresydra

También en el siglo III a.C y por otro inventor griego conocido como Filón de Bizancio, se creó el que probablemente sea el primer robot de la historia, el *Sirviente Automático de Philo* [3]. El inventor y escritor recogió todas sus teorías en nueve libros. El número cinco, *Pneumatica* [4], describe el comportamiento del robot, capaz de llenar una taza de vino al poner el recipiente sobre la palma de la figura. En ese momento se articulan los mecanismos interiores, haciendo que los conductos se liberen y dejando así que el líquido fluya a través de la jarra.

Fig. 2.2. Sirviente Automático de Philo

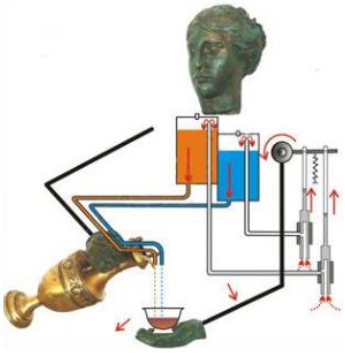


Fig. 2.3. Mecanismo interior

Avanzando en el tiempo, en 1495 Da Vinci creó en una de sus anotaciones un nuevo autómata, esta vez con un mecanismo mucho más complejo constituido a base de poleas y engranajes y fijado en el interior de una armadura, que la haría capaz de mover las extremidades y realizar actos como sentarse, levantar la visera y mover la cabeza. Aunque sólo se encontraron los escritos en el año 1950, se cree que Da Vinci pudo haber construido un prototipo de dicho autómata mientras trabajaba para el Duque de Milán. Estos descubrimientos inspiraron a la NASA para diseñar sus propios robots humanoides, creando en el año 2002 una versión a partir de los dibujos del inventor con Mark Rosheim al cargo, obteniendo como resultado un autómata completamente funcional. [5]

Fig. 2.4. Modelo de autómata de Da Vinci

Sin embargo, no todos los desarrollos trataron sobre autómatas humanoides. Siglos más adelante, en el año 1898, Nikola Tesla presentó en Nueva York el primer modelo de vehículo por control remoto, al que nombró *Teleautomaton*. Dicho artilugio consistía en un pequeño barco radio control que, a través de unas antenas instaladas en su superficie, obedecía a ordenes como avanzar, retroceder, girar o encender y apagar las bombillas. Este invento fue patentado por el mismo creador, describiendo así el primer mecanismo de control remoto para cualquier tipo de vehículo. [6]

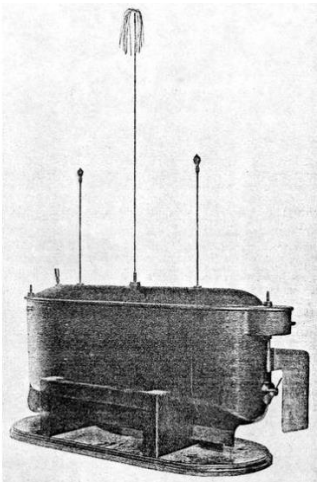


Fig. 2.5. Teleautomaton [6]

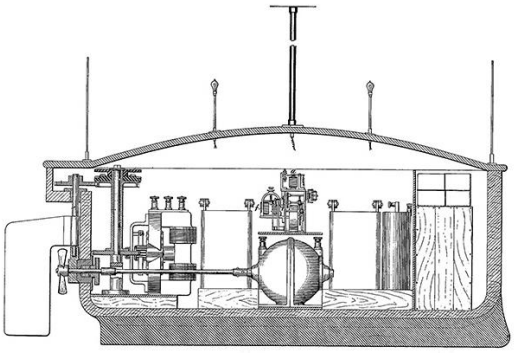


Fig. 2.6. Interior del Teleautomaton [6]

En el ámbito industrial hubo que esperar hasta el año 1961, momento en el que se instaló el primer robot industrial por General Motors. Llamado *Unimate*, este robot contaba con un brazo articulado y comandos almacenados en un tambor magnético, lo que le permitía el transporte y soldado de piezas en los automóviles, acelerando y facilitando así la cadena de montaje. [7]

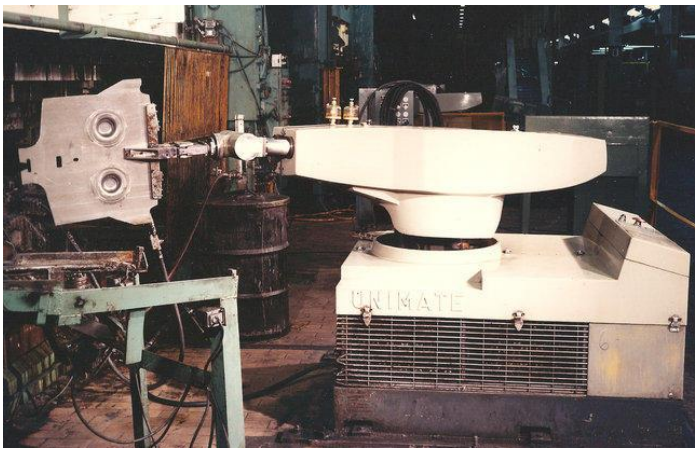


Fig. 2.7. Unimate [8]

En esa misma década, pocos años después, el Instituto de Investigación de Stanford dio a luz al primer robot móvil capaz de realizar toma de decisiones, que recibió el nombre de *Shakey* [9]. Dotado con sensores para interactuar con el entorno y gracias a su sistema basado en acciones y metas, era capaz de realizar maniobras tales como navegar de un punto a otro, encender y apagar las luces y manipular objetos. El robot fue programado en lenguaje *LISP (LISt Processor)* y disponía de una lista de acciones a desarrollar, ordenadas por medio de un plan. Estos fueron los primeros pasos de lo que hoy se conoce como la Programación Automática, utilizada también en el presente trabajo.

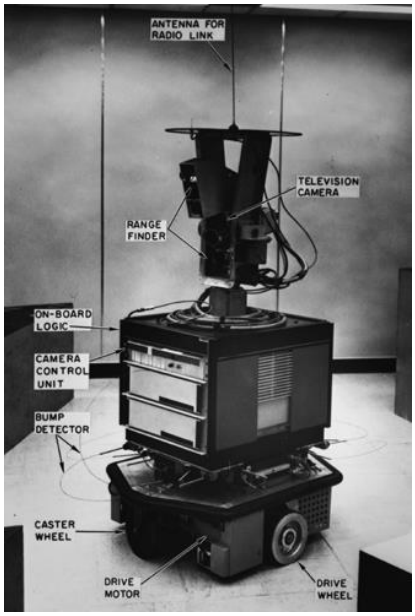


Fig. 2.8. Robot Shakey [10]

Como se podrá ver en una imagen posterior, dicho robot es similar al que se destina el software desarrollado en este trabajo, el modelo Pioneer 3 DX. Desde ese momento y hasta nuestros días se han realizado importantes avances en el ámbito de la robótica, mezclándose en numerosas ocasiones con la inteligencia artificial. Hasta el momento, los robots se pueden diferenciar en cuatro generaciones [11]:

- **Primera generación**: los denominados robots manipuladores. Constan de un mecanismo sencillo y no reciben ningún tipo de información del entorno.
- **Segunda generación**: o también llamados robots de aprendizaje. Son capaces de recibir cierta información de su entorno y almacenarla en un dispositivo junto con instrucciones. Normalmente son utilizados para repetir una secuencia de movimientos realizada por un ser humano.
- **Tercera generación**: se trata de los robots con control sensorizado. En este caso el controlador es un computador que analiza la información recibida por los sensores y resuelve la siguiente acción a ejecutar. En esta generación comienza a utilizarse la Inteligencia Artificial.

- **Cuarta generación**: llamada robots inteligentes, surge a partir de la generación anterior, pero añadiendo nuevos avances como el control sobre el estado del proceso, que permite mejorar la ejecución de las tareas en tiempo real mediante estrategias más complejas.

El robot en cuestión sobre el que se destina el desarrollo es un tipo de cuarta generación utilizado para exploración. En estos casos el dispositivo debe estar preparado para lidiar con un entorno desconocido, por lo que es importante la información recibida por parte de los sensores y la forma de tratarla y diseñar un plan en función a ello.

En este campo son numerosos los avances realizados, siendo probablemente el más conocido el *Rover* utilizado para misiones espaciales por su capacidad para moverse sobre superficies planetarias [12]. Dichos robots deben poseer alto grado de modularidad tanto en hardware como en software, para facilitar así su reconfiguración y mantenimiento en caso de que fuese necesario.

Fig. 2.9. Rover Curiosity [13]

En este tipo de robots es importante también la creación de código modular y reutilizable, de forma que se incremente el conocimiento sobre el dominio del problema y se creen algoritmos genéricos capaces de dar respuesta a problemas recurrentes en distintos sistemas, ahorrando así recursos útiles para nuevos desarrollos. Según el *Research Institute for Advanced Computer Science (RIACS)* [14], una colaboración entre *Universities Space Research Association (USRA)* [15] y *NASA Ames Research Center* [16] en una publicación realizada acerca de la estructura software de este tipo de vehículos [17], esta se trata de una de las mayores dificultades a la hora del desarrollo del rover, ya que el software integra numerosas disciplinas y es largo y complejo, además de verse limitado por el hardware, que no es el mismo en todas las ocasiones. Además, debe tratarse de un modelo que se desarrolle en tiempo real, haciendo que todos los módulos necesarios se encuentren operativos en el momento preciso. Por ello uno de los grandes objetivos es la creación de un *framework* que pueda funcionar de

forma similar en diferentes mecanismos aplicando los patrones de diseños apropiados para el dominio, generalmente mediante la división en pequeños módulos que permita una mejora y simplificación de la integración de las funcionalidades.

2.2. Planificación automática

La planificación automática es una rama de la Inteligencia Artificial que estudia la creación automatizada de planes, es decir, la capacidad de generar una secuencia de acciones de forma que estas resuelvan un problema. Usualmente se aplica para controlar agentes inteligentes, tales como robots o vehículos no tripulados. Para ello es necesario proporcionar una serie de información mediante una definición formal. La más común se trata de PDDL (*Planning Domain Definition Language)*, un lenguaje basado en predicados desarrollado en 1998. Mediante este lenguaje se pueden especificar las características del problema a través de predicados y algo realmente útil como son las métricas, beneficiosas cuando se desea que el problema se resuelva atendiendo a unas condiciones, como puede ser que se utilice la menor cantidad de combustible posible en el dominio de un vehículo [18]. Entonces, utilizando esta definición es necesario representar los siguientes elementos:

- **Acciones**: tareas a ejecutar por el agente. Cada una de estas tareas tiene unas precondiciones para que pueda realizarse y unos efectos. Por ejemplo, en el dominio de un rover, si la tarea es navegar desde *plataforma1* a *plataforma3*, la precondición será que el vehículo se encuentre en la plataforma uno y el efecto será que su posición se actualizará a la plataforma tres. Como es de suponer, las precondiciones y efectos serán más numerosas cuanto más complejo sea el dominio.

- **Estados**: para la correcta ejecución del problema es necesario proporcionar el estado inicial. Este es dado mediante un conjunto de predicados, por ejemplo un estado inicial puede ser definido como *(at rover1 plataform3)*, que indica que inicialmente el rover uno está situado en la plataforma 3. Los estados intermedios se van generando durante la ejecución del programa mediante la realización de acciones. Estos estados son utilizados para las métricas o funciones de evaluación y para las metas.

- **Metas**: se trata del estado final que debe alcanzar el problema. Al igual que los estados, se describen en forma de predicado. Siguiendo el mismo hilo, si la meta es *(at rover1 plataform5)* el sistema deberá ejecutar un plan para que, mediante una serie de pasos, el rover uno acabe situado en la plataforma cinco. El objetivo del planificador es hallar dicha secuencia de pasos.

El estado inicial y las metas conforman el problema, mientras que las acciones (con sus precondiciones y efectos) y la definición de los predicados genéricos forman el dominio. De esta manera, para un mismo dominio se pueden tener *n* problemas, siendo así una buena forma de aprovechar conocimiento [19].

Aunque hoy PDDL y la planificación automática han evolucionado bastante y cuentan con un gran potencial, los inicios no fueron muy distintos a lo que hoy en día se conoce. Como se ha mencionado en el apartado anterior, *Shakey* fue concebido a finales de los años 60 por el Instituto de Investigación de Stanford como el primer robot capaz de "razonar" sobre sus propias acciones. Bien, pues esto fue gracias a STRIPS (*STandford Research Institute Problem Solver),* el primer software de planificación automática, basado del mismo modo en acciones, estados y metas.

A partir de ese punto el lenguaje fue evolucionando para dar lugar a *ADL (Action Description Language)* en 1987, que entre otras mejoras introdujo los literales y disyunciones negativas. [20]

Fue entonces cuando surgió PDDL en 1998 como inspiración de estos dos lenguajes y tomándolos como parte de su sistema junto con otros lenguajes de representación, lo que hizo posible el desarrollo de la Competición Iternacional de Planicación (International Planning Competition, IPC) en el marco de la *International Conference on Automated Planning and Scheduling (ICAPS)* [21] y donde PDDL ha ido evolucionando hasta la fecha. Los objetivos de ICAPS son la difusión e innovación en el campo de la Planificación Automática y el *Scheduling* mediante una serie de actividades, entre las que se incluye la citada competición internacional.

2.3. Arduino y Raspberry

Tal como indican en su propia página web, "Arduino es una plataforma electrónica de código abierto basada en hardware y software fácil de usar. Utiliza el lenguaje de programación Arduino (basado en cableado) y el software Arduino (IDE), basado en procesamiento". [22]

Utilizado hoy en día para el desarrollo de dispositivos digitales o aplicaciones interactivas, surgió en el año 2003 como una herramienta dirigida a que los estudiantes dieran sus primeros pasos en electrónica y programación. Tras su comercialización en 2005 ha comenzado a ser ampliamente utilizado por su bajo coste y por tratarse de un dispositivo multiplataforma, pudiendo utilizar su software en Windows, Linux y Mac. Existe además gran variedad de modelos en función de las necesidades de cada usuario,

diferenciando por ejemplo si van a ser utilizadas para el Internet de las Cosas o para fines educativos.

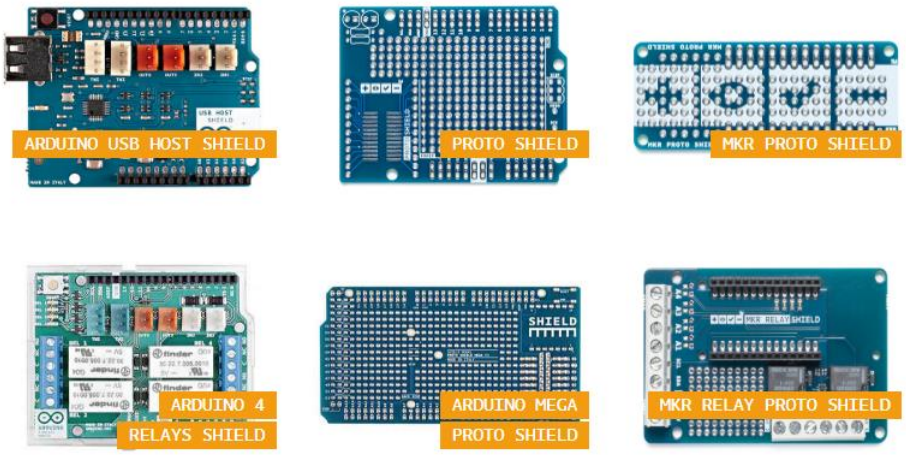


Fig. 2.10. Modelos Arduino [23]

En cuanto a la Raspberry Pi, se trata de un computador de tamaño reducido creado por la Fundación Raspberry Pi, comprometida con la educación de jóvenes y adultos en el ámbito de la informática y las nuevas tecnologías. [24]

Conectada a un monitor y un teclado es capaz de realizar las mismas funciones que cualquier ordenador, además de tener capacidad para interactuar con el exterior y conectar numerosos dispositivos, resultado práctica para gran número de proyectos. Su software es de código abierto y cuenta con un sistema operativo oficial para todos los modelos de Raspberry, en concreto una versión de Debian llamada *Raspbian*, aunque actualmente se encuentran disponibles para su descarga imágenes adaptadas de otros sistemas operativos [25]. Para ejecutarlo en el dispositivo es necesaria una tarjeta SD en la que introducir la imagen del sistema.

Cuenta con varios modelos, entre los que únicamente se han ido realizando mejoras, no habiendo modelos específicos según el uso al que se destine. El último de ellos se trata de la Raspberry Pi 3 Model B+, distribuido en marzo de 2018 [26]. Esta placa se trata de un modelo bastante más novedoso que Arduino, ya que fue lanzado al mercado en febrero de 2012, colapsando los servidores de ventas a su salida.

Fig. 2.11. Raspberry Pi [26]

A simple vista pueden parecer muy similares, pero hay diferencias notables entre ellos. Mientras que Arduino es un microcontrolador, Raspberry se trata de una versión reducida de un ordenador completo. El primero en su versión nativa no soporta multitarea ni multihilo, por lo que resulta idóneo para tareas que no requieran potencia de cómputo y deban ejecutarse durante un largo periodo de tiempo, como por ejemplo algunos proyectos de electrónica que involucran leds. Esto proporciona gran rapidez de ejecución, junto con el hecho de que nada más conectarla comienza a ejecutar la tarea programada.

La desventaja de la Raspberry es que necesita un sistema operativo para poder utilizarla, pero a cambio se gana en potencia de cómputo y memoria, haciéndola conveniente en proyectos más complejos o que requieran multitarea.

2.4. PELEA

PELEA, acrónimo de *Planning, Execution and LEarning Architecture*, se trata de una herramienta creada por varias universidades españolas, entre ellas la Universidad Carlos III de Madrid, para la integración de planificación, control y aprendizaje automático en robots, capaz de monitorizar la ejecución del plan para un mayor seguimiento del problema.

Dicha aplicación consta de dos niveles, siendo el alto nivel las acciones relacionadas directamente con el planificador, y el bajo nivel vinculado a las tareas a desarrollar en el robot.

En su versión más sencilla, que es la usada en este trabajo, PELEA está formado por tres módulos principales que se ejecutan de forma paralela, explicados a continuación.

Módulo de Ejecución

Encargado de realizar los pasos fundamentales para el funcionamiento de la herramienta. Al inicio recibe el problema a resolver a alto nivel, relativo al dominio del problema que utilizará el planificador introducido, en el que se establece el estado inicial y los objetivos finales. Mediante la clase *HightToLow* obtiene también la conversión de las acciones a bajo nivel necesarias para la comunicación con el robot. De la misma forma, recibe la información enviada a bajo nivel por el robot, que es enviada al módulo de monitorización. Mediante la comunicación con el resto de módulos, recibe los parámetros adecuados para continuar el ciclo de ejecución.

Módulo de Monitorización

La información generada en el Módulo de Ejecución se envía a este componente, que se encarga en primer lugar de comprobar si las metas se han cumplido. En caso de ser así significaría el fin del problema. En caso contrario, Monitorización comprueba que el estado a bajo nivel sea el correcto. Como la información es recibida a bajo nivel, es necesario llamar al componente *LowToHigh* para obtener el equivalente a alto nivel, y que la información pueda seguir siendo manejada por el sistema. Para evitar la monitorización de todas las variables, el Módulo de Soporte a la Decisión le indica cuáles son aquellas que debe comprobar, es decir, aquellas que afecten al plan en cada momento. Si el estado es el correcto continúa la ejecución, en caso contrario la información es enviada a Soporte a la Decisión para la replanificación.

Tras la actualización del plan, este es enviado de nuevo al módulo de Ejecución, que toma la primera acción del plan y la ejecuta.

Módulo de Soporte a la Decisión

Sus principales funciones son decidir sobre qué variables deben ser monitorizadas y cuáles son sus rangos válidos y replanificar en caso de que sea necesario. Esta tarea la realiza llamando al Planificador de Alto Nivel, un módulo adicional que incluye un planificador automático. Si el error hace el plan inválido tendrá que decidir si crear un nuevo plan o si el estado actual se puede corregir.

Este proceso se repite de forma cíclica hasta que se cumplen las metas especificadas en el problema. Entonces, de forma general el sistema se puede resumir en el siguiente esquema:

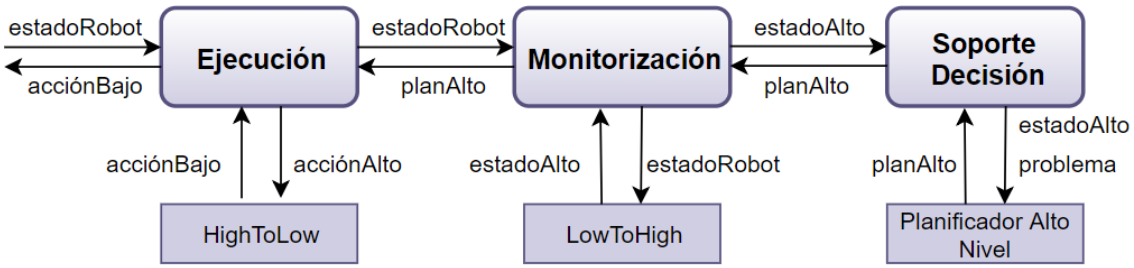


Fig. 2.12. Esquema de ejecución de PELEA

PELEA conoce en todo momento la acción a realizar y sus efectos, por lo que sabe de antemano cuál es el estado esperado. Durante el envío de mensajes entre los distintos módulos se monitoriza el estado actual frente al esperado, permitiendo detectar diferencias entre ambos. En caso de hallar un resultado no esperado, el sistema es capaz de replanificar para encontrar un plan alternativo que solucione el estado encontrado. El encargado de dicha tarea es el componente de Soporte a la Decisión, que determinará cuáles son las nuevas acciones a ejecutar o si los cambios no son relevantes y se puede continuar con el plan inicial.

Se puede encontrar información adicional sobre PELEA en el documento referenciado [27].

2.5. ROS

En su versión general, ROS (*Robot Operative System)* se trata de un framework para el desarrollo de software a implementar en robots. Permite a su vez el control del robot de una forma relativamente fácil, mediante comandos o instrucciones gracias a un conjunto de librerías y herramientas, que incluye tareas tales como navegación, mapeo, localización, diagnóstico y paso de mensajes. [28]

Las herramientas más comunes para transmisión de mensajes en este ámbito son los *topics* o los servidores. El primer caso consiste en un método muy flexible de publicadores y subscriptores anónimos, basado en nodos y mensajes, de forma que el robot transmite de forma continua información acerca de su estado y aquellos nodos que desean conocer estos datos simplemente se suscriben al *topic* para recibir dicha información. [29]

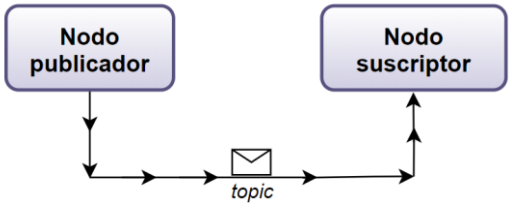


Fig. 2.13. Estructura nodos

Por ejemplo, un robot podría tener varios *topics*, y en cada uno de ellos estaría publicando su velocidad, posición y demás sensores conectados que generen datos. Si el sistema de control quiere conocer estos datos tendría que suscribirse a dichos *topics* y extraer de ellos la información deseada.

Sin embargo, esta arquitectura no es válida en el caso de que se necesite un sistema de petición-respuesta, como es el caso de este trabajo. Para ello existe un segundo método basado en la técnica de cliente-servidor [30], que facilita en gran medida la comunicación para el caso planteado. Dicho sistema consiste en un servidor, en este caso el robot, que debe encontrarse continuamente disponible para cualquier solicitud que le realice el cliente.

Por su parte el cliente, en este caso el sistema desarrollado, se encargará de enviar dentro del mensaje de petición la información a gestionar por parte del servidor, que a su vez enviará un mensaje de respuesta cuando finalice su tarea. Este bucle se repetirá hasta que alguno de los dos decida cortar el servicio, aunque el servidor puede seguir ejecutándose sin que haya ningún cliente conectado, pero no al revés.

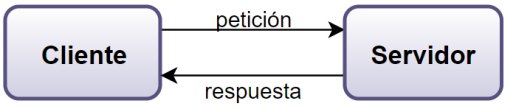


Fig. 2.14. Estructura cliente-servidor

Ya que ambas partes necesitan intercambiar información sólo en momentos puntuales, este método ahorra el envío y la lectura continua de datos que en su mayoría serían descartados. ROS posee más funcionalidades y librerías que permiten el control del robot en distintos aspectos, algunas de las cuales se mencionarán más adelante.

2.6. Trabajos similares

Con la tecnología citada anteriormente, son muchas las personas que han decidido crear su propio robot, por diversión o con fines de aprendizaje e investigación. A continuación se muestran algunos ejemplos, de los que se detallarán sus principales características.

Basado en Arduino

Como ya se ha comentado, Arduino cuenta con algunas limitaciones en cuanto a cómputo y multitarea, por lo que los proyectos relacionados con vehículos móviles son curiosos y una buena forma de aprender sobre la tecnología y de la que se puede sacar provecho, pero entre ellos no se encuentran grandes sistemas ni agentes 100% inteligentes, sino que la mayoría son controlados de forma remota.

Al buscar sobre ello se pueden encontrar algunos proyectos definidos como "robots autónomos", pero entrando en detalle se puede descubrir que se trata de vehículos que esquivan objetos o "sigue-líneas".

Para el primero de ellos se ha elegido como ejemplo un Trabajo Fin de Grado de Ingeniería Informática, desarrollado en la Universidad de Valencia [31]. El cometido del robot es circular en línea recta hasta que detecte un obstáculo. En ese momento deberá girar hacia el lado contrario del que ha sido localizado el objeto y continuar moviéndose hacia adelante hasta que vuelva a darse la misma situación. En el caso de girar y encontrarse con un nuevo objeto, seguirá girando en el mismo sentido hasta que nada le bloquee el paso. En este caso también se añade la complejidad del montaje del robot.

En cuanto al "sigue-líneas", se trata de un robot típico que cuenta con tutoriales para su construcción [32]. Dotado con unos sensores infrarrojos, es capaz de seguir una línea que contraste con la superficie en la que se coloca. Si los sensores están colocados sobre la línea el vehículo continúa recto, en caso de que uno de ellos se sitúe fuera, el robot girará para volver a colocar el sensor sobre la raya. En el caso de que ambos sensores salgan de su trayectoria el vehículo parará.

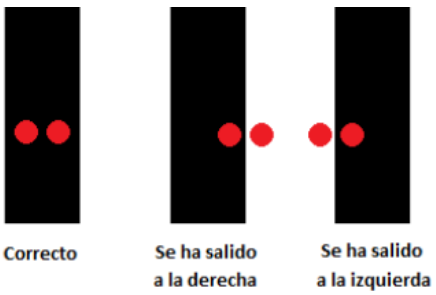


Fig. 2.15. Sensores robot "sigue-líneas" [32]

Estos son dos de algunos de los proyectos, se pueden encontrar más ejemplos en la siguiente referenciada [33].

Por lo tanto, durante la investigación no se ha encontrado ningún sistema creado con Arduino que cuente con planificación automática.

Basado en Raspberry Pi

Gracias a las mayores prestaciones de la Raspberry, esta es utilizada para proyectos que requieren más capacidad, si bien también es elegida trabajos similares a los realizados con Arduino. Sin embargo, se puede observar que se introducen nuevas herramientas, como ROS (*Robot Operating System)*, un framework para el desarrollo de software robótico cuyos detalles se explican en el punto 4.3.

Ejemplos de ello son dos robots móviles, uno de ellos un robot simple manejado mediante comandos [34] y el otro un robot de limpieza *roomba* modificado para seguir las instrucciones enviadas a través de la interfaz gráfica de ROS. [35]

Con dicho *framework* es posible convertir cualquier hardware de un robot genérico en un robot fácilmente controlable. Otro ejemplo del uso de dicha herramienta se trata de este proyecto [36] con un robot móvil Pioneer3AT, un modelo similar al utilizado en este trabajo. Por tanto, ROS parece una opción viable para su uso debido a sus buenos resultados y a su fácil integración para ese tipo de hardware.

Así pues, se puede concluir que en el campo de la robótica móvil la Raspberry Pi proporciona mayor rango de oportunidades, permitiendo la unión de distintas herramientas. Pese a ello, tampoco se ha hallado ningún trabajo que relacione la Raspberry Pi con Planificación Automática.

3. OBJETIVOS Y ALCANCE

Una vez estudiada la situación actual de las tecnologías disponibles y los trabajos ya desarrollados sobre el tema, se establecen los objetivos a cumplir en el trabajo.

La finalidad de este trabajo es la creación de software para el control de un robot P3-DX mediante el uso de una Raspberry Pi y planificación automática.

Para ello se parte desde dos sistemas elegidos por su potencial: ROS y PELEA. Tal como se ha explicado, el primero de ellos puede permitir el control de un robot mediante comandos y envío de mensajes, mientras que el segundo se trata de un software de planificación y ejecución.

El objetivo entonces es, mediante la unión de ambas herramientas, crear un sistema para el control autónomo de un robot e integrarlo en una Raspberry Pi para una mayor facilidad de uso. Dicho sistema será capaz de ejecutar un plan en el dominio de un robot Pioneer 3 DX y enviar las acciones secuencialmente a través de una interfaz que permite la conversión entre alto y bajo nivel.

Este objetivo se puede dividir en las siguientes fases:

- Investigación sobre ROS: para implementar primero una solución basada en este *framework* es necesario conocer primero sus características y las posibilidades que ofrece, para decidir cuál de ellas es la más adecuada. Una vez realizado también hay que adquirir los conocimientos prácticos para su manejo.

- Investigación sobre PELEA: al tratarse de una arquitectura compleja con varios módulos, en primer lugar hay que comprender su ciclo de ejecución y observar de forma práctica los puntos clave en los que el sistema puede ser modificado.

- Integración de ROS con PELEA: si el robot contase únicamente con ROS, este podría ser manejado de forma remota. Para dotarle de autonomía es necesario realizar la unión entre esta herramienta y PELEA, de forma que las acciones decididas por este último módulo sean enviadas al robot mediante ROS.

- Creación del dominio y problemas para el robot P3-DX: en función de las tareas que pueda realizar el vehículo, se implementará un dominio con las acciones y predicados adecuados que permita la resolución de escenarios.

- Creación de la interfaz entre el alto y el bajo nivel: El robot no puede recibir las acciones directamente tal como las determina PELEA, sino que tienen que ser pasadas antes a bajo nivel. Para ello hay que definir un tipo de mensaje y realizar la conversión a datos comprensibles por el vehículo.

- Carga en Raspberry Pi: una vez que el sistema sea funcional, se procederá a cargarlo en la Raspberry Pi.
- Realización de pruebas: una vez establecido el sistema se deben realizar las pruebas necesarias para comprobar que los resultados son los esperados.

Esto se realizará tratando de crear un sistema lo más modular y reutilizable posible, de forma que resulte escalable a otros sistemas similares mediante simples cambios o que, ante algún cambio en el robot, como puede ser añadir alguna nueva tarea o una modificación en los datos del problema, esto se resuelva en el sistema con la menor cantidad de esfuerzo y tiempo posible.

4. MATERIALES Y MÉTODOS

En este apartado se detallan las herramientas utilizadas y el porqué de su elección, así como las alternativas de diseño.

4.1.Elección de las herramientas

Para el desarrollo del trabajo se han necesitado una serie de herramientas base, entre las que se encuentran:

- PELEA (*Planning, Execution and Learning Architecture)*
- RosJava
- Raspberry Pi 3 Modelo B
- Robot Pioneer P3-DX
- Eclipse

Se detallan a continuación, indicando su funcionamiento y características y las ventajas que aporta su elección al desarrollo del trabajo.

4.2. PELEA

Esta herramienta ya ha sido presentada en la sección acerca del estado del arte.

Una de las ventajas que influyó en gran medida en su elección es que se trata de un sistema con gran potencial, pero a su vez es independiente del dominio y problema introducido, siempre y cuando el lenguaje de los mismos sea entendido por el planificador. Con PDDL esto no supone ningún problema, ya que se trata de un lenguaje estandarizado para la definición de dominios que aceptan la mayoría de planificadores. En este caso PELEA cuenta con el planificador Metric-FF [37], una extensión del planificador FF (*Fast-Forward)* [38] independiente del dominio adaptada para manejar variables numéricas.

Para adaptar esta herramienta al robot seleccionado habrá que realizar modificaciones en ciertos puntos, que se detallan en el apartado *6.3*, donde se muestra cómo se ha configurado para este trabajo.

4.3.Rosjava

Habiendo introducido ya ROS en la sección del estado del arte, en este punto se hablará sobre su versión utilizada para implementar el sistema.

Para la creación de la estructura del cliente-servidor explicada en el punto 2.5 se ha utilizado RosJava, una variante de ROS escrita en Java, utilizada comúnmente para el desarrollo de aplicaciones Android. Pese a no tratarse de ese tipo de trabajo, se ha decidido la implementación en dicho lenguaje debido al conocimiento previo del mismo, facilitando así el proceso de desarrollo, y en mayor medida a causa de que PELEA está creado también en Java. Como el principal propósito del trabajo es integrar ambas herramientas, la opción de RosJava se trataba claramente de la más compatible a la hora de realizar dicha tarea.

La versión utilizada ha sido RosJava Kinetic [39] frente a Indigo [40], debido a que es la última liberada y cuenta con tutoriales actualizados. También, en el caso de una instalación normal de ROS, Kinetic sólo soporta Ubuntu 16.04, el Sistema Operativo instalado en el ordenador de desarrollo y en la placa Raspberry Pi, mientras que Indigo está ideada para versiones anteriores. Por tanto, pareció más adecuado optar por la versión más actual.

4.4. Raspberry

Tras comentar las diferencias entre Arduino y Raspberry, para este trabajo se ha decidido emplear la Raspberry Pi, para asegurar así que durante el trabajo no se encuentren limitaciones debido a rendimiento.

El tipo utilizado es la Raspberry Pi 3 Modelo B, del año 2016. La razón de usar este modelo y no el B+, su versión mejorada, es que este nuevo modelo se comercializó en marzo 2018, por lo tanto para ese momento ya se había realizado la elección de materiales y las mejoras no son tan significativas como para que mereciese la pena la compra de una nueva placa.

Las diferencias más notables entre ambos modelos se encuentran resumidas en la siguiente tabla:

Raspberry Pi	Procesador	RAM	Bluetooth	Wifi
Modelo B	1.2 GHz	1 GB	4.1	2.4GHz
Modelo B+	1.4 GHz	1 GB	4.2	2.4GHZ y 5GHz

Tabla 4.1. Comparación modelos Raspberry Pi [41]

Como se puede comprobar, en lo que afecta a este trabajo, el modelo B+ sólo supondría algo más de velocidad, por lo que la Raspberry Pi 3 Modelo B es perfectamente válida para este desarrollo.

Adicionalmente, también ha sido necesario el uso de una tarjeta de memoria SD para cargar el sistema operativo en la Raspberry Pi, para lo que se ha elegido el modelo *Toshiba microSDHC UHS-I Card. 32 GB. Clase 10.*

4.5. El robot

Se trata de un Pioneer 3 DX, una base robótica perteneciente a MobileRobot.

En su versión más simple consta de un sonar frontal, una batería, ruedas, un microcontrolador y el software Pioneer SDK, que proporciona un conjunto de librerías y herramientas para el control del vehículo, ayudando así en el desarrollo de los proyectos.

Fig. 4.1. Robot Pioneer 3 DX [42]

Debido a su facilidad y durabilidad, dicho robot está principalmente destinado a fines docentes o de investigación. El sistema viene ya armado e integrado y cuenta con gran cantidad de documentación y soporte técnico, de forma que cualquier problema puede quedar resulto en corto espacio de tiempo. Es resistente también a impactos y debido a su pequeña dimensión, de 45x38x23cm, resulta ideal para espacios cerrados como clases o laboratorios.

Este robot no cuenta con ningún ordenador integrado por lo que es común el uso del P3DX con un ordenador portátil conectado y apoyado en la superficie para su control, con la desventaja de que se pierde así espacio para colocar otros sensores o accesorios, entre los que se encuentran mapeo y visión, manipulación o audio. La alternativa elegida en este trabajo ha sido una placa Raspberry Pi o un clúster de placas conectadas

al robot que realicen los cómputos, ahorrando así espacio para la conexión de complementos en el caso de querer ampliar el sistema.

Fig. 4.2. Robot P3-DX con altavoces [42]

Fig. 4.3. Robot P3-DX con brazo [42]

Relacionado también con el trabajo, el robot Pioneer 3 DX es una buena opción ya que dentro del paquete SDK se encuentra ARIA, un *framework* para el control de los datos generados por el robot. Gracias a esto se puede adaptar el vehículo para estructuras cliente-servidor, como es el caso. Además, ROS cuenta con un paquete llamado RosAria compatible con la biblioteca del robot, de forma que se facilita la creación de la estructura y la integración de ROS en el robot.

Se puede encontrar información adicional sobre el robot P3DX en la guía [42] o en el manual de operación [43].

4.6. Eclipse

Se trata de un IDE (*Integrated Development Enviroment)* de programación ampliamente utilizado para la programación en Java, desarrollado por IBM en 2001 hasta el año 2004, momento en el que se creó la Fundación Eclipse, que actúa hoy en día como administrador de la comunidad [44]. Su proyecto más conocido se trata de este IDE, que proporciona un editor de texto y un compilador en tiempo real, aunque estas herramientas también han sido extendidas en el mismo IDE para otros lenguajes como C o C++. Por ello, Eclipse no cuenta con todas las funciones como base, sino que dispone de *plugins* a través de los cuales el usuario puede agregar las funciones que sean necesarias.

Eclipse cuenta con numerosas versiones desde su fecha de lanzamiento, concretamente una por año desde 2004. Al tratarse de una herramienta de programación basada en Java, para su funcionamiento requiere tener previamente instalado el JDK (*Java Development Kit).*

Existen alternativas de uso con otros IDEs de programación, pero, aunque no existen estadísticas oficiales sobre su uso, se han realizado algunos estudios cuyos resultados apuntan a que Eclipse es el más empleado por los desarrolladores.

A través de *Google Trends*, en el año 2015 se obtuvieron los siguientes resultados [45] en tanto por ciento basados en las búsquedas realizadas de cada uno de los entornos:

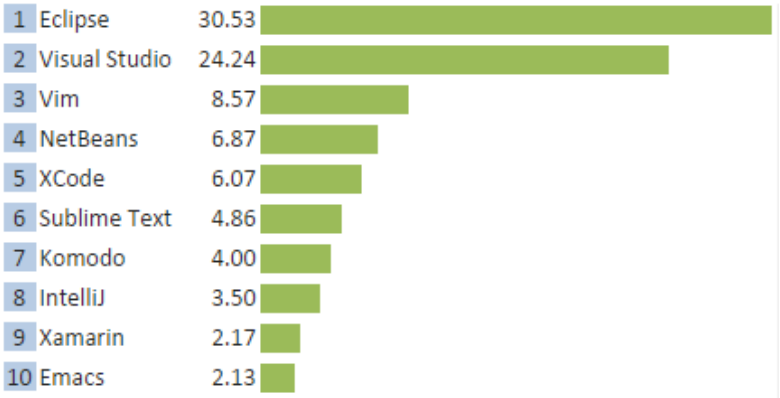


Fig. 4.4. IDEs más buscados según Google Trends [45]

Un año antes, en 2014, se realizó una encuesta a los usuarios de *Codeanywhere* [46], una plataforma para edición de código online, para conocer qué entornos de trabajo usaban en su versión de escritorio bajo la pregunta "¿Qué aplicación utilizan para el desarrollo, además de *Codeanywhere*?". Los resultados obtenidos fueron los siguientes:

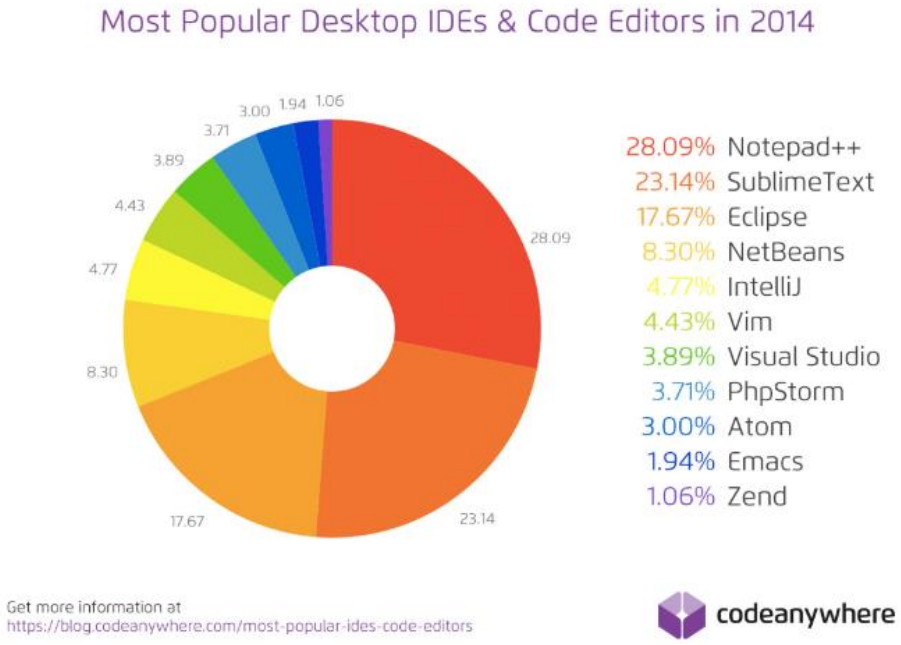


Fig. 4.5. Estudio de Codeanywhere sobre aplicaciones de desarrollo [46]

Como se puede ver, *Notepad++* y *SublimeText* lideran sobre Eclipse, pero estos son sólo editores de texto, por lo que no se pueden considerar como IDEs.

En base a estos datos y debido a que ya se contaba con conocimiento previo de la herramienta, se ha decidido el uso de Eclipse como entorno de programación del trabajo, ahorrando así un coste adicional de aprendizaje, además de ser una herramienta cuyo funcionamiento cubría completamente las necesidades del trabajo, facilitando la corrección de errores gracias a la compilación en tiempo real y el sistema de *debug*. Por ello, se optó desde el principio por su uso.

4.7. Alternativas de diseño

Existen varias herramientas como opción alternativa a ROS. Algunas de ellas son:

- ***Mobile Robot Programming Toolkit (MRPT)*** [47]: ofrece un conjunto de librerías útil para el desarrollo y la investigación en el campo de la robótica. Se trata de un software de código abierto escrito en C++. Incluye también una sección de planificación, pero está destinada en su mayoría al cálculo de rutas, no se trata de una planificación de tareas.

- ***Microsoft Robotics Developer Studio*** [48]: se trata de un entorno de programación y simulación de robots basado en Windows 7, pudiendo ser utilizado tanto por programadores profesionales como aficionados. Su lenguaje principal de programación es C# y cuenta con numerosas versiones, siendo la última *Robotics Developer Studio 4*, cuya fecha de publicación fue en el año 2012 y no se actualiza desde entonces.

- ***RoboComp*** [49]: proporciona un marco de trabajo para el desarrollo de software, que permiten el intercambio de información con el robot. Se trata de una herramienta de código abierto cuya programación se realiza en C++ o Python. Sin embargo, este método cuenta con poca documentación y su instalación sólo ha sido probada hasta Ubuntu 15.10.

Por estos motivos la mejor opción parece ser ROS, que cuenta también con una comunidad activa y un foro de discusión destinado a dudas [50].

Una vez decidido el uso de ROS, otra alternativa hubiese sido el uso de *topics* para la comunicación, pudiendo acceder así directamente a información de sensores o cualquier información que el robot publicase. Esto habría supuesto estar continuamente leyendo la información recibida para saber cuándo el robot ha terminado de ejecutar o en qué estado se encuentra. Por ese motivo, para el objetivo perseguido se decidió que era más práctico el uso de un cliente-servidor.

Durante el desarrollo del proyecto se han ido planteando más alternativas de diseño. Algunas ya se han comentado en este apartado 4 y el resto se detallarán más adelante, en el punto 6.

5. ENTORNO SOCIO-ECONÓMICO

Durante los últimos años se han realizado grandes avances en el mundo de la robótica, haciéndose cada vez más visible en el día a día de la sociedad. Tecnologías en auge como el Internet de las Cosas, el Aprendizaje Automático, drones, impresoras 3D, robots asistenciales, de uso doméstico o médico están creando ya un nuevo concepto llamado "Cuarta Revolución Industrial" o "Revolución Robótica".

Típicamente la robótica o automatización ha sido utilizada para la realización industrial de tareas simples o repetitivas, pero hoy en día, gracias a la Inteligencia Artificial y a la creciente capacidad de gestión de datos, o *Big Data,* esta está siendo aplicada a otro tipo de tareas. De esta forma surgen los vehículos móviles autónomos, como el *Cruise AV* de General Motors [51], o los capaces de analizar resultados de pruebas genéticas, como *Watson* de IBM [52]. Aunque en algunos proyectos aún queda mucho trabajo por delante, en otros ya se obtienen resultados satisfactorios, como es el caso del rover de exploración en Marte, capaz de cumplir sus metas mediante técnicas de planificación automática.

El ritmo de estos avances es tan rápido que hace que uno de los mayores debates sociales sobre ello sea qué número de trabajos hoy en día realizados por personas serán llevados a cabo por máquinas en un futuro. No hay duda de que la creciente innovación tecnológica tendrá cada vez mayor impacto en la sociedad y para intentar predecir en qué medida se han realizado numerosos estudios. Sin embargo, no resulta un análisis sencillo, ya que la mayoría de los puestos de trabajo no están compuestos de una única tarea, ya que muchos de ellos incluyen también habilidades sociales, etc. Para un examen más preciso, habría que analizar entonces qué tareas son susceptibles de automatizarse.

Dicho estudio fue realizado por profesores de la Universidad de Oxford en su trabajo "*The future of employment: how susceptible are Jobs to computerisation?*" [53], en el que se diferencian tres tareas que por el momento no pueden ser realizadas por agentes inteligentes:

- **Tareas de percepción y manipulación**: se han realizado avances en cuanto a que un robot identifique objetos por sus formas básicas, pero no resulta tan simple cuando se le sitúa en un ambiente que requiere tareas de percepción más complejas, como puede ser el interior de una casa, donde es difícil establecer algún tipo de estructura. En estos mismos entornos también se complica la manipulación de objetos, que tienden a ser irregulares, ya que aún no se ha conseguido alcanzar el nivel humano de aptitud para este tipo de habilidades.

- **Inteligencia creativa**: existe ya software que hace posible la creación de artilugios surgidos del ingenio, ya que existen robots capaces de realizar obras artísticas, como AARON de Harold Cohen [54], capaz de crear pinturas originales. Pero existen valores creativos que son difícilmente programables. Por ejemplo, para que un ser

humano haga una broma interviene gran cantidad de conocimiento, además del contexto en el que se encuentra. Resulta complicado, entonces, incluir en una máquina tal información y que ella sea capaz de hacer combinaciones de ideas que den lugar al ingenio o la intuición.

- **Inteligencia social**: en cuestiones de interacción humana aún hay bastante trabajo por delante. Sobre este tema, el Test de Turing es una de las pruebas más conocidas, en la que una máquina intenta emular un comportamiento inteligente similar al de un ser humano mediante una conversación por medio textual. El problema es que la inteligencia social es importante en tareas de negociación y persuasión, campos que todavía escapan a la tecnología actual.

Según otro estudio efectuado por CaixaBank siguiendo estos criterios, se puede obtener la siguiente conclusión:

> *"Como hemos señalado, se estima que la tecnología ya es capaz de automatizar profesiones cualificadas (véase el riesgo que corren contables, analistas financieros y economistas), mientras que aquellas en las que la interacción humana y la creatividad tienen más importancia (médicos de familia, músicos) son las que están más protegidas. (...) Científicos (creatividad) y gerentes (interacción social) tienen poco riesgo, mientras que los administrativos se concentran en el grupo de alto riesgo."* [55]

En el aspecto económico, con algo de investigación se puede observar que hay diferentes teorías y estudios al respecto. Cuando la robótica esté completamente instaurada en las compañías, se producirá una disminución de coste y un aumento de la producción, lo que significará la devaluación del producto. Las suposiciones que acompañan a estos hechos apuestan por que el desempleo será total, mientras que otros defienden que sólo quedarán trabajos en el área tecnológica, creciendo tanto la demanda en este sector que los salarios caerán bruscamente.

Lo cierto es que actualmente la tecnología está siendo adaptada por las empresas si es barata y pueden obtenerse beneficios de ella, siendo la gran barrera los costes de instalación y mantenimiento, mientras que en Japón está ayudando frente a la escasez de trabajadores, cubriendo dicho hueco de mercado [56].

Como dato final que apoya que la "revolución robótica" está cerca, un informe publicado por Transparency Market Research [57], en el año 2016 la robótica mundial recaudó un total de treinta y cinco mil millones de dólares, en torno a una cuarta parte de lo que se espera para el año 2025, con una suma de ciento cuarenta y siete mil millones de dólares.

Respecto al trabajo descrito en este documento, no se espera obtener ningún beneficio de él, ya se ha sido desarrollado sólo como Trabajo Fin de Grado para implementarlo en

un robot cuyo uso, tal como se ha explicado anteriormente, está principalmente destinado a fines docentes o de investigación.

5.1.Marco regulador

Este trabajo se encuentra sujeto a la licencia Creative Commons Reconocimiento – No comercial – Sin Obra Derivada.

Adicionalmente no está bajo restricciones legales, ya que actualmente no hay ninguna regulación que pueda aplicarse al ámbito de la robótica.

Sí cabe destacar que ante el auge de este campo, el 16 de febrero de 2017 fue aprobado por el Parlamento Europeo un informe con recomendaciones destinadas a la Comisión sobre normas de Derecho civil sobre robótica [58]. En él se sugieren distintos puntos sobre ética, responsabilidad, seguridad, registro de robots y demás cuestiones jurídicas que regularán el futuro de la robótica y la inteligencia artificial.

6. DISEÑO E IMPLEMENTACIÓN

En este apartado de explica el proceso de desarrollo del sistema, desde el planteamiento de los casos de uso y la especificación de requisitos hasta la unión de todos los elementos para formar el programa.

6.1. Casos de uso

Una vez fijado el alcance el sistema y teniendo claros los componentes principales que intervendrán en él, se detallan los diferentes casos de uso que pueden presentarse durante el empleo del sistema.

Los casos de uso pueden resumirse en el siguiente esquema:

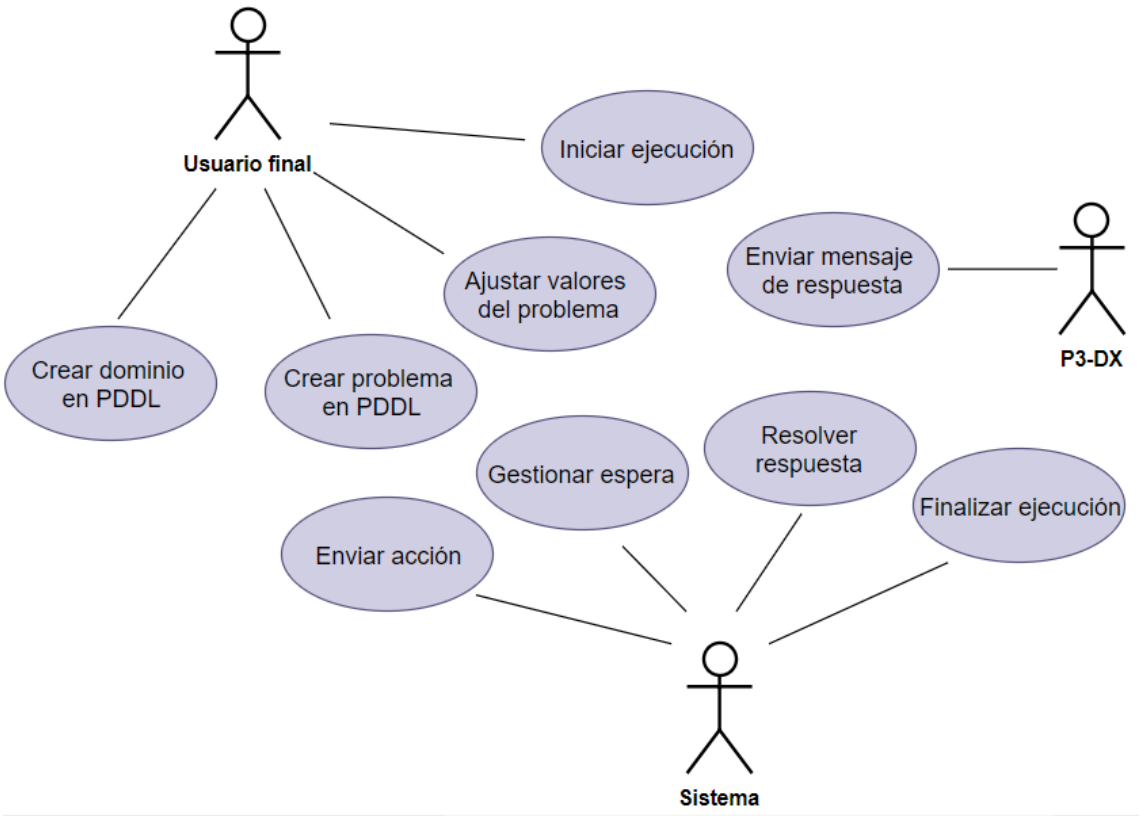


Fig. 6.1. Esquema de casos de uso

Donde pueden distinguirse los siguientes actores:

- Usuario final: Persona que hará uso del sistema implementado.
- P3-DX: Robot Pioneer 3 DX.
- Sistema: Módulo desarrollado de planificación y envío de mensajes.

Para la especificación se utilizará una tabla como la siguiente por cada caso de uso. Los atributos se encuentran definidos bajo ella.

Nombre		Código	CU-XX
Actor			
Objetivo			
Precondiciones			
Efectos			
Proceso			

Tabla 6.1. Modelo de tabla casos de uso

- **Nombre**: Breve descripción única sobre el caso planteado.
- **Código**: numeración unívoca que representa el caso de uso, donde XX es la identificación numérica en el rango [01,99].
- **Actor**: tipo de usuario que interactúa con el sistema.
- **Objetivo**: finalidad perseguida por el actor al ejecutar la acción.
- **Precondiciones**: circunstancias que deben darse para poder cumplir el objetivo.
- **Efectos**: resultados de la realización de la acción si se cumplen las precondiciones.
- **Proceso**: pasos a dar para realizar la operación.

Se definen entonces los siguientes casos de uso:

Nombre	Crear dominio en PDDL.	Código	CU-01
Actor	Usuario final.		
Objetivo	Implementar un dominio (acciones y estados posibles) en PDDL.		
Precondiciones	El usuario debe tener conocimientos de PDDL		
Efectos	Se tiene un fichero *.pddl* con el dominio generado.		
Proceso	1. El usuario analiza las características del dominio deseado y lo modela en PDDL. 2. Tiene en cuenta los posibles estados mediante predicados y las precondiciones y efectos de cada acción.		

Tabla 6.2. CU-01

Nombre	Crear problema en PDDL.	Código	CU-02
Actor	Usuario final.		
Objetivo	Implementar un problema a resolver sobre el dominio en PDDL.		
Precondiciones	El dominio tiene que haber sido creado.		
Efectos	Se tiene un fichero *.pddl* con el problema generado.		
Proceso	1. El usuario identifica cuál es el problema a resolver 2. Haciendo uso de los predicados y acciones del dominio, modela el problema en PDDL. 3. Proporciona como mínimo un estado inicial y una meta.		

Tabla 6.3. CU-02

Nombre	Ajustar valores del problema.	Código	CU-03
Actor	Usuario final.		
Objetivo	Determinar los valores numéricos asociados al problema.		
Precondiciones	Dominio y problema tienen que haber sido creados.		
Efectos	Es posible enviar al robot la información adecuada.		
Proceso	1. Observar los atributos creados en el problema. 2. Para cada uno de ellos, el usuario debe asignar su valor correspondiente. 3. Dicho valor se incluye en un fichero de configuración.		

Tabla 6.4. CU-03

Nombre	Iniciar ejecución.	Código	CU-04
Actor	Usuario final.		
Objetivo	Lanzar el sistema para su comunicación con el robot.		
Precondiciones	- El robot debe estar encendido. - El dominio y el problema deben haber sido creados. - Se han establecido los valores numéricos para el problema.		
Efectos	Comienza la ejecución del sistema.		
Proceso	1. Encender el robot. 2. Iniciar la parte cliente. 3. Iniciar el sistema de planificación automática.		

Tabla 6.5. CU-04

Nombre	Enviar acción	Código	CU-05
Actor	Sistema		
Objetivo	Mandar al robot la siguiente acción a ejecutar.		
Precondiciones	- El robot debe estar disponible. - El sistema ha sido lanzado. - La acción ha sido convertida a bajo nivel.		
Efectos	Se envía la acción y el sistema queda suspendido a la espera de respuesta.		
Proceso	1. PELEA selecciona la siguiente tarea a realizar. 2. Se transforma dicha tarea a bajo nivel. 3. ROS monta el mensaje de petición y lo envía al robot.		

Tabla 6.6. CU-05

Nombre	Gestionar espera.	Código	CU-06
Actor	Sistema.		
Objetivo	Hacer que el sistema quede suspendido tras el envío del mensaje e impedir que el sistema espere un tiempo prolongado por un mensaje de respuesta en el caso de que este no llegue.		
Precondiciones	- La acción ha tenido que ser enviada en un mensaje de petición. - El sistema ha quedado detenido. - Se ha cumplido el tiempo establecido como *time out*		
Efectos	El sistema se detiene, mostrando el error detectado o se reanuda la ejecución.		
Proceso	1. El sistema envía la acción y permanece a la espera de respuesta. 2. En caso de recibir respuesta reanuda la ejecución. 3. En caso de superar el tiempo de espera se detiene totalmente la ejecución.		

Tabla 6.7. CU-06

Nombre	Enviar mensaje de respuesta	Código	CU-07
Actor	P3-DX		
Objetivo	El robot responde al sistema al finalizar la tarea.		
Precondiciones	La acción ha sido enviada por parte del cliente.		
Efectos	El sistema de control gestiona la respuesta recibida por parte del servidor.		
Proceso	1. La parte del servidor crea un mensaje de respuesta con el resultado de la acción (erróneo o correcto). 2. Se envía mediante ROS.		

Tabla 6.8. CU-07

Nombre	Resolver respuesta	Código	CU-08
Actor	Sistema		
Objetivo	El sistema actúa acorde a la respuesta recibida por parte del servidor.		
Precondiciones	El robot ha enviado un mensaje de respuesta.		
Efectos	El sistema de control continúa con la ejecución normal o replanifica en caso de error.		
Proceso	1. Se evalúa la respuesta recibida por parte del robot. 2. En caso de ser positiva continúa con la ejecución normal del plan. 3. En caso de ser negativa se replanifica.		

Tabla 6.9. CU-08

Nombre	Finalizar ejecución	Código	CU-09
Actor	Sistema		
Objetivo	Detener el sistema una vez cumplidas las metas.		
Precondiciones	Se han alcanzado las metas establecidas en la definición del problema.		
Efectos	El sistema se detiene, mostrando un resumen de los resultados.		
Proceso	1. Se evalúa el estado actual del sistema. 2. Se comprueba que el estado corresponde al objetivo establecido. 3. Se detiene totalmente el sistema.		

Tabla 6.10. CU-09

6.2. Requisitos

La especificación de requisitos es una parte fundamental del desarrollo del sistema, ya que sienta las bases sobre las que se trabajará y establece los objetivos principales que tiene que cumplir el software.

Los requisitos se detallarán mediante la siguiente tabla:

Código		Fuente	
Prioridad	☐ Alta ☐ Media ☐ Baja	Claridad	☐ Alta ☐ Media ☐ Baja
Necesidad	☐ Esencial ☐ Deseable ☐ Opcional		
Verificabilidad	☐ Alta ☐ Media ☐ Baja	Estabilidad	☐ Estable ☐ Inestable
Descripción			

Tabla 6.11. Modelo de tabla requisitos

- **Código**: cada tipo de requisito tendrá su propio código, a distinguir entre RF para requisitos funcionales y RNF para requisitos no funcionales. Cada uno de ellos irá seguido de su número en orden secuencial de 00 a 99.
- **Fuente**: se señala el origen del requisito. Este puede haber surgido de un caso de uso o haber sido pensado por el desarrollador.
- **Prioridad**: se elegirá la preferencia del requisito entre alta, media o baja. Se considera alta cuando sea una función básica del sistema.
- **Claridad**: cada requisito debe tener una definición única y no ambigua. Si algún requisito tuviese una claridad media o baja, debería ser modificado.
- **Necesidad**: si se trata de un requisito básico del sistema, este debe ser implementado y marcado como esencial. En otro caso podría ser negociable.
- **Verificabilidad**: indica en qué grado es posible la comprobación del requisito. Si tiene alta verificabilidad se trata de que el requisito forma parte de las funciones básicas del sistema. En otro caso, podría tratarse de una implementación que cubra algún tipo de caso que sea complejo reproducir o requiera de mayores conocimientos.
- **Estabilidad**: se trata de la posibilidad de cambio del requisito durante el proyecto. Si aún está sujeto a modificaciones será necesario marcarlo para realizar un seguimiento del mismo.
- **Descripción**: funcionalidad a cumplir o restricción impuesta del requisito.

6.2.1. Requisitos funcionales

Describen la funcionalidad que tiene el software, es decir, las acciones que puede realizar. Derivan de los casos de uso.

Código	RF-01	Fuente	CU-01
Prioridad	☒ Alta ☐ Media ☐ Baja	Claridad	☒ Alta ☐ Media ☐ Baja
Necesidad	☒ Esencial ☐ Deseable ☐ Opcional		
Verificabilidad	☒ Alta ☐ Media ☐ Baja	Estabilidad	☒ Estable ☐ Inestable
Descripción	El sistema podrá leer dominios implementados en PDDL.		

Tabla 6.12. RF-01

Código	RF-02	Fuente	CU-02
Prioridad	☒ Alta ☐ Media ☐ Baja	Claridad	☒ Alta ☐ Media ☐ Baja
Necesidad	☒ Esencial ☐ Deseable ☐ Opcional		
Verificabilidad	☒ Alta ☐ Media ☐ Baja	Estabilidad	☒ Estable ☐ Inestable
Descripción	El sistema será capaz de resolver problemas codificados en PDDL asociados a un dominio.		

Tabla 6.13. RF-02

Código	RF-03	Fuente	CU-03
Prioridad	☒ Alta ☐ Media ☐ Baja	Claridad	☒ Alta ☐ Media ☐ Baja
Necesidad	☒ Esencial ☐ Deseable ☐ Opcional		
Verificabilidad	☒ Alta ☐ Media ☐ Baja	Estabilidad	☒ Estable ☐ Inestable
Descripción	El sistema podrá convertir los valores y acciones de alto nivel a bajo nivel.		

Tabla 6.14. RF-03

Código	RF-04	Fuente	CU-03
Prioridad	☒ Alta ☐ Media ☐ Baja	Claridad	☒ Alta ☐ Media ☐ Baja
Necesidad	☒ Esencial ☐ Deseable ☐ Opcional		
Verificabilidad	☒ Alta ☐ Media ☐ Baja	Estabilidad	☒ Estable ☐ Inestable
Descripción	El sistema podrá leer los ficheros de configuración introducidos.		

Tabla 6.15. RF-04

Código	RF-05	Fuente	CU-03
Prioridad	☒ Alta ☐ Media ☐ Baja	Claridad	☒ Alta ☐ Media ☐ Baja
Necesidad	☒ Esencial ☐ Deseable ☐ Opcional		
Verificabilidad	☒ Alta ☐ Media ☐ Baja	Estabilidad	☒ Estable ☐ Inestable
Descripción	En caso de haberlos, el sistema mostrará información sobre errores de formato en los ficheros de configuración.		

Tabla 6.16. RF-05

Código	RF-06	Fuente	CU-04
Prioridad	☒ Alta ☐ Media ☐ Baja	Claridad	☒ Alta ☐ Media ☐ Baja
Necesidad	☒ Esencial ☐ Deseable ☐ Opcional		
Verificabilidad	☒ Alta ☐ Media ☐ Baja	Estabilidad	☒ Estable ☐ Inestable
Descripción	Una vez iniciada la ejecución esta deberá mantenerse hasta la resolución del problema sin intervención del usuario.		

Tabla 6.17. RF-06

Código	RF-07	Fuente	CU-04
Prioridad	☒ Alta ☐ Media ☐ Baja	Claridad	☒ Alta ☐ Media ☐ Baja
Necesidad	☒ Esencial ☐ Deseable ☐ Opcional		
Verificabilidad	☒ Alta ☐ Media ☐ Baja	Estabilidad	☒ Estable ☐ Inestable
Descripción	El cliente contactará con el servidor adecuado, mostrando un error en caso contrario.		

Tabla 6.18. RF-07

Código	RF-08	Fuente	CU-05
Prioridad	☒ Alta ☐ Media ☐ Baja	Claridad	☒ Alta ☐ Media ☐ Baja
Necesidad	☒ Esencial ☐ Deseable ☐ Opcional		
Verificabilidad	☒ Alta ☐ Media ☐ Baja	Estabilidad	☒ Estable ☐ Inestable
Descripción	Se deberá enviar al robot la siguiente acción a realizar según el orden establecido en el plan.		

Tabla 6.19. RF-08

Código	RF-09	Fuente	CU-06
Prioridad	☒ Alta ☐ Media ☐ Baja	Claridad	☒ Alta ☐ Media ☐ Baja
Necesidad	☒ Esencial ☐ Deseable ☐ Opcional		
Verificabilidad	☒ Alta ☐ Media ☐ Baja	Estabilidad	☒ Estable ☐ Inestable
Descripción	Una vez enviada la acción el sistema se mantendrá suspendido hasta recibir el mensaje de respuesta por parte del servidor, continuando la ejecución posteriormente.		

Tabla 6.20. RF-09

Código	RF-10	Fuente	CU-06
Prioridad	☒ Alta ☐ Media ☐ Baja	Claridad	☒ Alta ☐ Media ☐ Baja
Necesidad	☒ Esencial ☐ Deseable ☐ Opcional		
Verificabilidad	☒ Alta ☐ Media ☐ Baja	Estabilidad	☒ Estable ☐ Inestable
Descripción	El cliente contará con un tiempo limitado de espera por la respuesta del servidor.		

Tabla 6.21. RF-10

Código	RF-11	Fuente	CU-06
Prioridad	☒ Alta ☐ Media ☐ Baja	Claridad	☒ Alta ☐ Media ☐ Baja
Necesidad	☒ Esencial ☐ Deseable ☐ Opcional		
Verificabilidad	☒ Alta ☐ Media ☐ Baja	Estabilidad	☒ Estable ☐ Inestable
Descripción	El sistema parará la ejecución y cortará la comunicación con el servidor en caso de superar el tiempo de espera establecido, mostrando el error.		

Tabla 6.22. RF-11

Código	RF-12	Fuente	CU-07
Prioridad	☒ Alta ☐ Media ☐ Baja	Claridad	☒ Alta ☐ Media ☐ Baja
Necesidad	☒ Esencial ☐ Deseable ☐ Opcional		
Verificabilidad	☒ Alta ☐ Media ☐ Baja	Estabilidad	☒ Estable ☐ Inestable
Descripción	El sistema recibirá la respuesta enviada desde el robot.		

Tabla 6.23. RF-12

Código	RF-13	Fuente	CU-08
Prioridad	☒ Alta ☐ Media ☐ Baja	Claridad	☒ Alta ☐ Media ☐ Baja
Necesidad	☒ Esencial ☐ Deseable ☐ Opcional		
Verificabilidad	☒ Alta ☐ Media ☐ Baja	Estabilidad	☒ Estable ☐ Inestable
Descripción	El sistema continuará con el plan de ejecución normal en caso de recibir *feedback* positivo por parte del robot.		

Tabla 6.24. RF-13

Código	RF-14	Fuente	CU-08
Prioridad	☒ Alta ☐ Media ☐ Baja	Claridad	☒ Alta ☐ Media ☐ Baja
Necesidad	☒ Esencial ☐ Deseable ☐ Opcional		
Verificabilidad	☒ Alta ☐ Media ☐ Baja	Estabilidad	☒ Estable ☐ Inestable
Descripción	En caso de recibir una respuesta errónea del robot, el sistema replanificará las acciones a realizar.		

Tabla 6.25. RF-14

Código	RF-15	Fuente	CU-09
Prioridad	☒ Alta ☐ Media ☐ Baja	Claridad	☒ Alta ☐ Media ☐ Baja
Necesidad	☒ Esencial ☐ Deseable ☐ Opcional		
Verificabilidad	☒ Alta ☐ Media ☐ Baja	Estabilidad	☒ Estable ☐ Inestable
Descripción	El proceso acabará siempre con el robot en modo teleoperación, para que pueda seguir siendo controlado mediante comandos en caso de ser necesario.		

Tabla 6.26. RF-15

Código	RF-16	Fuente	CU-09
Prioridad	☒ Alta ☐ Media ☐ Baja	Claridad	☒ Alta ☐ Media ☐ Baja
Necesidad	☒ Esencial ☐ Deseable ☐ Opcional		
Verificabilidad	☒ Alta ☐ Media ☐ Baja	Estabilidad	☒ Estable ☐ Inestable
Descripción	El sistema parará la ejecución y cortará la comunicación con el servidor en caso de cumplir todas sus metas, mostrando un resumen sobre el proceso.		

Tabla 6.27. RF-16

6.2.2. Requisitos no funcionales

Los requisitos no funcionales describen las características requeridas por el sistema o durante el proceso de desarrollo. A diferencia de los requisitos funcionales, que determinan capacidad, estos marcan restricciones.

Código	RNF-01	Fuente	Desarrollador
Prioridad	☒ Alta ☐ Media ☐ Baja	Claridad	☒ Alta ☐ Media ☐ Baja
Necesidad	☒ Esencial ☐ Deseable ☐ Opcional		
Verificabilidad	☒ Alta ☐ Media ☐ Baja	Estabilidad	☒ Estable ☐ Inestable
Descripción	Se requiere de JDK versión 7 o superior.		

Tabla 6.28. RNF-01

Código	RNF-02	Fuente	Desarrollador
Prioridad	☒ Alta ☐ Media ☐ Baja	Claridad	☒ Alta ☐ Media ☐ Baja
Necesidad	☒ Esencial ☐ Deseable ☐ Opcional		
Verificabilidad	☒ Alta ☐ Media ☐ Baja	Estabilidad	☒ Estable ☐ Inestable
Descripción	El sistema operativo de desarrollo debe ser Ubuntu 16.04.		

Tabla 6.29. RNF-02

Código	RNF-03	Fuente	Desarrollador
Prioridad	☒ Alta ☐ Media ☐ Baja	Claridad	☒ Alta ☐ Media ☐ Baja
Necesidad	☒ Esencial ☐ Deseable ☐ Opcional		
Verificabilidad	☒ Alta ☐ Media ☐ Baja	Estabilidad	☒ Estable ☐ Inestable
Descripción	Los repositorios de Ubuntu deben estar configurados para permitir *restricted*, *universe* y *multiverse*.		

Tabla 6.30. RNF-03

Código	RNF-04	Fuente	Desarrollador
Prioridad	☒ Alta ☐ Media ☐ Baja	Claridad	☒ Alta ☐ Media ☐ Baja
Necesidad	☒ Esencial ☐ Deseable ☐ Opcional		
Verificabilidad	☒ Alta ☐ Media ☐ Baja	Estabilidad	☒ Estable ☐ Inestable
Descripción	Durante la ejecución del sistema *roscore* debe estar también ejecutando.		

Tabla 6.31. RNF-04

Código	RNF-05	Fuente	Desarrollador
Prioridad	☒ Alta ☐ Media ☐ Baja	Claridad	☒ Alta ☐ Media ☐ Baja
Necesidad	☒ Esencial ☐ Deseable ☐ Opcional		
Verificabilidad	☒ Alta ☐ Media ☐ Baja	Estabilidad	☒ Estable ☐ Inestable
Descripción	Eclipse debe incluir las siguientes variables de entorno: - ROS_ROOT - ROS_PACKAGE_PATH - PYTHONPATH - PATH		

Tabla 6.32. RNF-05

Código	RNF-06	Fuente	Desarrollador
Prioridad	☒ Alta ☐ Media ☐ Baja	Claridad	☒ Alta ☐ Media ☐ Baja
Necesidad	☒ Esencial ☐ Deseable ☐ Opcional		
Verificabilidad	☒ Alta ☐ Media ☐ Baja	Estabilidad	☒ Estable ☐ Inestable
Descripción	El lenguaje de implementación debe ser Java.		

Tabla 6.33. RNF-06

Código	RNF-07	Fuente	Desarrollador
Prioridad	☒ Alta ☐ Media ☐ Baja	Claridad	☒ Alta ☐ Media ☐ Baja
Necesidad	☒ Esencial ☐ Deseable ☐ Opcional		
Verificabilidad	☒ Alta ☐ Media ☐ Baja	Estabilidad	☒ Estable ☐ Inestable
Descripción	La comunicación entre cliente y servidor debe hacerse por medio de una interfaz personalizada que establezca los atributos necesarios.		

Tabla 6.34. RNF-07

Código	RNF-08	Fuente	Desarrollador
Prioridad	☒ Alta ☐ Media ☐ Baja	Claridad	☒ Alta ☐ Media ☐ Baja
Necesidad	☒ Esencial ☐ Deseable ☐ Opcional		
Verificabilidad	☒ Alta ☐ Media ☐ Baja	Estabilidad	☒ Estable ☐ Inestable
Descripción	Dominios y problemas deben ser modelados en PDDL.		

Tabla 6.35. RNF-08

Código	RNF-09	Fuente	Desarrollador
Prioridad	☒ Alta ☐ Media ☐ Baja	Claridad	☒ Alta ☐ Media ☐ Baja
Necesidad	☒ Esencial ☐ Deseable ☐ Opcional		
Verificabilidad	☒ Alta ☐ Media ☐ Baja	Estabilidad	☒ Estable ☐ Inestable
Descripción	Los ficheros de configuración deberán escribirse con una determinada estructura, finalizando cada línea con *;*		

Tabla 6.36. RNF-09

Código	RNF-10	Fuente	Desarrollador
Prioridad	☒ Alta ☐ Media ☐ Baja	Claridad	☒ Alta ☐ Media ☐ Baja
Necesidad	☒ Esencial ☐ Deseable ☐ Opcional		
Verificabilidad	☒ Alta ☐ Media ☐ Baja	Estabilidad	☒ Estable ☐ Inestable
Descripción	Cada tarea a realizar por el robot llevará asociado un código, especificado en el fichero de configuración.		

Tabla 6.37. RNF-10

Código	RNF-11	Fuente	Desarrollador
Prioridad	☒ Alta ☐ Media ☐ Baja	Claridad	☒ Alta ☐ Media ☐ Baja
Necesidad	☒ Esencial ☐ Deseable ☐ Opcional		
Verificabilidad	☒ Alta ☐ Media ☐ Baja	Estabilidad	☒ Estable ☐ Inestable
Descripción	Para la resolución del problema en PDDL, será necesario como mínimo especificar el estado inicial y un objetivo a conseguir.		

Tabla 6.38. RNF-11

6.3. Matriz de trazabilidad

Para comprobar que cada caso de uso queda cubierto por al menos un requisito, se ha elaborado la siguiente matriz de trazabilidad. En ella se pueden observar todas las relaciones.

	CU-01	CU-02	CU-03	CU-04	CU-05	CU-06	CU-07	CU-08	CU-09
RF-01	X								
RF-02		X							
RF-03			X						
RF-04			X						
RF-05			X						
RF-06				X					
RF-07				X					
RF-08					X				
RF-09						X			
RF-10						X			
RF-11						X			
RF-12							X		
RF-13								X	
RF-14								X	
RF-15									X
RF-16									X

Tabla 6.39. Matriz de trazabilidad requisitos - casos de uso

6.4. Desarrollo

En los siguientes apartados se detalla cómo se ha realizado la integración de todas las herramientas con el fin de cubrir todos los casos de uso que pueden plantearse y teniendo en cuenta los requisitos definidos.

El proceso de desarrollo pasó por la carga de PELEA en el IDE de trabajo, para importar posteriormente en la misma herramienta el proyecto cliente-servidor y realizar así la integración de ambos sistemas. El punto de unión entre ellos es la interfaz de comunicación, que hubo que implementar a través de una estructura que da forma a los mensajes que intercambian ambas partes.

Esto conllevó realizar algunas decisiones de diseño respecto a cómo ajustar esta interfaz para que encajase sin problemas y realizase su función correctamente.

Una vez realizado lo anterior, se pasó a la creación del dominio en PDDL mediante el modelado de los predicados y las acciones posibles que puede llevar a cabo el robot. En este punto se estudiaron las posibilidades de replanificación con las que podía contar el sistema, teniendo en cuenta la ausencia de datos de los sensores.

Finalmente, el sistema fue probado con una serie de problemas mostrados en el punto 7 e introducido en la Raspberry Pi.

6.4.1. Carga de PELEA

Existe la opción de ejecutar PELEA por terminal, de forma que lanzando un *script* se ejecutan los tres módulos y se puede observar cómo avanza el proceso. Esto resulta poco práctico, ya retrasaría la depuración a la hora de unirlo con ROS. Por lo tanto, se decide importar PELEA en Eclipse para facilitar el desarrollo.

Para su funcionamiento se crean tres copias de la clase *main* y se ejecuta pasándole a cada una de ellas uno de los tres módulos por parámetro. A partir de entonces el resultado es el mismo, abriendo tres consolas se puede ver el ritmo de ejecución de cada uno.

El dominio de pruebas utilizado ha sido el de un rover espacial, ya que se asemeja bastante a las acciones de navegación a probar en el robot P3-DX. Adicionalmente, se han añadido tareas nuevas como el encendido de los motores tanto al dominio como al problema.

6.4.2. Configuración del entorno

Preparar RosJava para su uso lleva un proceso más largo, para los que se han seguido tutoriales oficiales de su página web de instalación y configuración del entorno. El proceso se puede encontrar detallado en el Anexo B.

6.4.3. Estructura cliente-servidor

La parte del servidor será la parte a implementar en el robot, mientras que el cliente será nuestro sistema a desarrollar. Para comenzar y comprobar el funcionamiento se creó la estructura con un ejemplo, disponible en el Anexo B, en el que el cliente solicita un array de cinco posiciones y cuando el servidor recibe la petición devuelve el listado de números. En este proceso intervienen dos elementos:

- **Mensaje de petición**: contiene la información que solicita el cliente. Esta petición será recibida por el servidor que tenga el mismo nombre que el cliente, por lo que hay que tener especial cuidado en enlazarlos correctamente. El servidor obtendrá los datos enviados y realizará la acción que se halla programado para ello.
- **Mensaje de respuesta**: Una vez que el servidor termine de realizar la tarea solicitada, creará un mensaje indicando una información que se encuentra esperando el cliente.

Esta estructura se define a través de una interfaz en un archivo *.srv* con la siguiente estructura (para el ejemplo dado):

```
int32[] size
---
int64[] res
```

Donde se indica el tipo de datos y nombres de las variables. La parte superior corresponde a la petición a realizar por el cliente y la inferior a la respuesta. Para adaptarlo al problema, una vez integrado habrá que personalizar el problema en función de la información a intercambiar con el robot.

6.4.4. Integración en Eclipse

Una vez importado PELEA en Eclipse, lo ideal es cargar también el cliente-servidor de RosJava en el mismo IDE para integrar así ambos proyectos. Para realizarse se encontró una forma mediante el uso de *gradle*, una herramienta que ayuda a la construcción de código. Cambiando algunos parámetros para añadir el *plugin* de Eclipse, detallados en el anexo B adjunto a este documento, y mediante un comando se obtuvo el cliente-servidor listo para incluirlo en dicho IDE.

Tras múltiples diseños y pruebas, se observó que la estructura con mejores frutos resultaba de insertar el cliente dentro del módulo de ejecución de PELEA. Dicho módulo cuenta con una clase propia, *ExecutionSqueleton* [sic], desde donde se llaman a ejecutar a las principales acciones. Esta clase cuenta con un gestor de mensajes, que redirige al método adecuado la comunicación recibida en *función* de la clase que sea.

Inicialmente se recibe el plan, que se pasa a gestionar para conseguir la siguiente acción a ejecutar. Con la acción ya seleccionada, se llama al método *executeAction*. Es posible que cada acción a alto nivel conlleve varias a bajo nivel, tales como encender dispositivos o realizar alguna comprobación dependiendo del dominio. Por lo tanto, en este método se llamará a la clase *high2low*, que haciendo uso del fichero de configuración con el mismo nombre obtendrá la acción o acciones a bajo nivel asociadas a la tarea a alto nivel a ejecutar.

A partir de este punto se ha modificado la funcionalidad del código, ya que cada una de las tareas tiene que ser enviada de forma independiente al robot. Es decir, cada tarea será enviada por el cliente con los parámetros adecuados, esperando posteriormente la confirmación de su realización.

Para ello se han creado los siguientes métodos y estructuras, donde se explica la justificación de diseño.

customProblem.cfg

Para el envío de las tareas al robot es necesario el uso de datos concretos, ya sean valores numéricos o booleanos, ya que la conversión de una tarea a alto nivel que indica que determinado vehículo navegue de un punto a otro, a bajo nivel se reduce a navegar a un *waypoint*. Si el robot recibe directamente la acción así convertida desconoce las coordenadas del lugar, por lo que no podrá completar la acción. De igual forma, el robot recibirá códigos de tareas en lugar de los nombres, por lo que habrá que realizar la conversión de la misma forma, enviando directamente el dígito correspondiente.

Como solución a ello se ha optado por incluir un nuevo fichero de configuración que contenga los valores correspondientes a cada una de las variables. Esto puede parecer una desventaja, ya que ahora se cuenta con dos *cfg*, el *HighToLow.cfg* mostrado en la

imagen anterior y este nuevo. Sin embargo, se ha decidido así porque dota de mayor reusabilidad al sistema.

Supongamos que para el mismo problema simplemente se quieren cambiar los puntos de navegación y la velocidad de desplazamiento. Entonces habría que ir al fichero *HighToLow* y cambiar dichas variables tantas veces como apareciesen, mientras que en este caso sólo habría que cambiar el valor una única vez para la variable elegida. De esta forma, se genera mayor independencia entre problema y datos de entrada. Por ejemplo, de otra forma, se podría añadir un fichero *customProblem.cfg* que contase con los datos de entrada de todas las posibles configuraciones. Si se introduce un problema que sólo utiliza la mitad de ellos no importa, ya que el sistema usará sólo los necesarios. El resto podrán mantenerse ahí para el caso de introducir un nuevo problema que sí haga uso de ellos.

Otra de las alternativas era introducir directamente la conversión y asociación de valores en el código, pero quedó descartada en el acto por tratarse de un método poco práctico y sin ningún tipo de abstracción.

El fichero *customProblem* tiene un estilo sencillo basado en la siguiente estructura:

```
<acción bajo nivel>: <código> ;
<variable con coordenadas>: (<x>,<y>) ;
<variable sin coordenadas> : <valor> ;
```

Donde cada línea debe comenzar con el nombre de la tarea o variable a bajo nivel, seguido de dos puntos y el valor dado. En caso de tratarse de coordenadas, estas deben estar encerradas entre paréntesis y separadas por una coma. Cada línea debe finalizar con un punto y coma. En el caso de no introducir los datos de esta manera, el sistema avisará sobre el error en el formato de entrada.

convertToSend()

Este método recibe la siguiente acción a ejecutar a bajo nivel y busca en *customProblem.cfg* el código asociado a dicha tarea. Para evitar errores adicionales de formato, se eliminan los comentarios y los espacios adicionales incluidos entre caracteres.

Posteriormente comprueba si la tarea tiene argumentos. En caso de tenerlos, continúa buscando en el *cfg* el valor asociado a cada uno de ellos. Una vez convertido, se guardarán en un *array* los valores de forma consecutiva. Por ejemplo, para la acción 'navegar' se tendría:

```
{<código>, <x>, <y>, <velocidad>}
```

De esta forma ya se tiene guardado en una misma lista toda la información a enviar al robot en el formato adecuado. Este *array* es devuelto por el método. En el siguiente apartado 6.4.5 se detallará este formato en las características de la interfaz de envío.

sendActionsToRobot()

Como el propio nombre indica, es el encargado de enviar las acciones al robot. Dentro de este método se ha embebido la parte fundamental del cliente de RosJava, por lo que en sus líneas se monta el mensaje para ser enviado al servidor. Para ello recibe por parámetro el *array* creado en el punto anterior e introduce los valores de acuerdo a la interfaz del mensaje creada.

Una vez realizado, se envía al servidor y se sigue ejecutando el código normal de PELEA. A continuación se encuentra el método *onSuccess*, que será llamado de forma automática cuando se reciba respuesta por parte del servidor, por lo que no será ejecutado acto seguido.

waiting()

En una ejecución aislada de ROS el sistema queda suspendido hasta que recibe la respuesta por parte del servidor, puesto que no tiene más código que ejecutar. Este no es el caso, ya que se encuentra situado dentro de PELEA, por lo que tras enviar el mensaje el sistema continúa con los siguientes pasos. Para evitar esta situación y seguir enviando acciones al robot antes de que haya finalizado las anteriores se ha implementado este método, de forma que el sistema queda suspendido hasta recibir contestación, momento en el que entra en *onSuccess* y gestiona la información recibida.

Para lograr esto se probaron soluciones relacionadas con la suspensión de hilos. En primer lugar se trató de usar los métodos *wait()* y *notify()*, que detiene la ejecución hasta que ocurre un determinado evento que vuelve a reanudarla, pero al hacerlo el sistema se detenía debido a una excepción de PELEA que no permitía realizar dicha acción. Esto puede deberse a que esta herramienta se basa en tres módulos que deben estar sincronizados, por lo que debe contar con alguna protección ante el uso de estos métodos para impedir la suspensión de un único módulo.

Ante este inconveniente se consideró el uso de métodos similares como *suspend()* y *resume()*, cuya finalidad es la misma que los mencionados, pero en este caso se trata de métodos obsoletos y no surtieron efecto.

Por lo tanto, debido a no encontrar otras alternativas adecuadas se optó por una solución sencilla basada en un *while* que mantiene el sistema detenido mientras no se reciba respuesta del servidor.

initClient()

En RosJava, el cliente es inicializado por una parte de la clase principal, *RosRun*. Para conseguir insertar el cliente dentro del módulo de ejecución se ha creado este apartado, que contiene el código necesario para su instanciación. De esta forma, al inicio de la clase *ExecutionSqueleton* se llama al método, logrando así que el cliente arranque junto con este módulo, simulando así ser la misma estructura.

Por tanto, gracias a estas estructuras se consigue introducir el cliente dentro del módulo de Ejecución, que recibe una a una las tareas del plan y las convierte a bajo nivel, para posteriormente consultar en el fichero de configuración los datos numéricos asociados a ella. Una vez que la acción se encuentra en el formato correcto esta se envía al robot, suspendiendo el sistema hasta recibir un mensaje de respuesta.

6.4.5. Interfaz de comunicación

La transmisión entre el cliente y el servidor está compuesta por dos tipos de mensajes. Estos formarán la interfaz de comunicación entre el sistema desarrollado y el robot, es decir, será el punto de unión entre ambos. Por lo tanto, la información debe ser transformada antes de su envío, ya que el robot está codificado a bajo nivel. La interfaz de comunicación supone entonces el paso entre alto a bajo nivel.

La estructura definida para la comunicación con el robot es la siguiente:

```
int64 codigo_tarea
string[] vector_mover
string[] vector_girar
string[] vector_x_y
---
string feedback
int64  eval
```

En el mensaje de respuesta se almacena la evaluación obtenida tras la realización de la tarea, pudiendo ser un 1 en caso satisfactorio o un 0 en caso de error. En la variable *feedback* se recibe una cadena de texto que indica la misma información a modo de *log.*

Respecto a la parte superior, el mensaje de petición, en el primer atributo se guarda el código asociado a la acción, y en los siguientes vectores se almacenan los parámetros necesarios para la realización de la tarea enviada. En función de la acción que a enviar se almacenará en el array correspondiente.

Las acciones que se pueden realizar son las siguientes:

- Activar motores: enciende los motores para que el robot pueda comenzar a moverse.
- Avanzar: dadas una velocidad, una distancia y la dirección, avanza en línea recta siguiendo dichos parámetros.
- Girar: realiza un giro para variar de dirección.
- Navegar: dado un punto en el espacio, el robot circula hasta él.
- Iniciar mapeo: para conocer el entorno es posible que el robot genere un mapa a través de la información recibida por los sensores.
- Finalizar mapeo: pone fin a la tarea anterior, obteniendo el mapa final.
- Modo teleoperación: se libera al robot del control automático, permitiendo que se dirija mediante instrucciones.

En la siguiente tabla se muestran las conversiones a bajo nivel:.

Acción	Código	Parámetros	Vector
Activar motores	0	-	
Avanzar	1	Velocidad, distancia, dirección	*vector_mover*
Girar	2	Velocidad, grados, dirección	*vector_girar*
Navegar	3	Coordenada *x*, coordenada *y*, grados	*vector_x_y*
Iniciar mapeo	4	-	
Finalizar mapeo	5	-	
Modo teleoperación	6	-	

Tabla 6.40. Envío de acciones al robot

Como se puede ver, hay acciones que no necesitan ningún tipo de información adicional. En estos casos bastará con enviar al robot el código de la tarea.

Creación del mensaje

Se encontró una forma simple de introducir el nuevo mensaje en la integración del sistema, que permitiría también la creación e inserción de nuevos tipos de interfaz en caso de ser necesario cambiar los atributos a intercambiar con el robot. Por ejemplo esto resultaría útil en el caso de que se añadiese al robot un nuevo sensor o dispositivo, como puede ser una cámara. Si se le ordena al robot la tarea de hacer una fotografía, esta probablemente tenga que ser enviada junto con parámetros acerca del punto del que se

desea tomar la imagen. En este caso, habría que añadir un nuevo vector al mensaje que contendrá la información necesaria en caso de realizar dicha tarea, por lo que bastaría con añadir el nuevo atributo a la estructura del mensaje y volver a compilar sólo esa parte.

Para ello, como ya se ha explicado, la estructura debe estar dentro de un archivo *.srv*, en este caso *ServerCustom.srv*. Una vez definido este archivo, mediante el uso de *genjava*, una herramienta que facilita en gran medida la creación de mensajes para su uso entre cliente y servidor, las clases necesarias para el uso de la interfaz, a diferenciar:

- *ServerCustom*: clase generada con la información contenida en el fichero *.srv*.
- *ServerCustomRequest*: clase que implementa los métodos *getters* y *setters* referidos a la parte del mensaje de petición. De esta forma se podrán establecer dichos parámetros para su envío y ser posteriormente leídos por el servidor.
- *ServerCustomResponse*: misma funcionalidad que la clase anterior, pero en este caso para los atributos de respuesta que irán en el mensaje de vuelta.

Con estas clases se crea automáticamente una librería, por lo que bastará con importarla al proyecto e indicar tanto en la parte del cliente como en la del servidor que ambos usarán mensajes de petición y respuesta del tipo *ServerCustomRequest* y *ServerCustomResponse*.

Una vez realizado, para crear el mensaje de envío se ha definido un objeto de tipo *ServerCustomRequest* a través del cual se pueden acceder a los *setters* para establecer los valores a enviar dentro del mensaje. Cuando los parámetros acordes a la tarea ya están fijados, el cliente envía el mensaje.

Interfaz alternativa

Se ha seguido este tipo de interfaz debido a que es la establecida para comunicarse con el robot. Sin embargo, esta conlleva un pequeño inconveniente, ya que dentro del objeto *ServerCustomRequest* se encuentran todos los *setters* involucrados en el mensaje de envío, a distinguir: *setVector_mover(), setVector_girar(), setVector_x_y()*. Por lo tanto, antes de crear el mensaje a enviar, habrá que ver qué tarea se está enviando y en función a ello rellenar alguno (o ninguno) de los arrays. Para hacer esto la única alternativa es introducir en el código condicionales para que rellene el *array* adecuado en función de la tarea a enviar, y una vez realizado, enviar el mensaje correcto.

Observando los vectores proporcionados, todos ellos son de tres posiciones, por lo que una mejor alternativa hubiese sido crear una lista genérica de dicho tamaño para el paso de mensajes. De esta forma se completaría el array genérico indistintamente de la tarea enviada sin tener que realizar más comprobaciones. Así se conseguiría una mayor abstracción en el sistema de control, dotándole de mayor independencia del problema. La gestión de la tarea y la información se realizaría entonces en la parte del servidor, propia del robot, que sí debe estar vinculada al dominio.

Otra forma de mejora hubiese sido incluir en el mensaje de respuesta información acerca de los sensores. PELEA es capaz de replanificar en caso de que el plan no siga su curso debido a algún error, pero para ello es necesario que conozca el estado actual, para así tener un nuevo punto de partida. Por ejemplo, se puede suponer un caso en el que al robot se le envía la tarea del navegar del *waypoint1*, situado en las coordenadas (1,4), al *waypoint2*, en el punto (5,4). En la interfaz actual, si a mitad de la tarea el robot se encuentra algún tipo de problema y no puede terminar la tarea, el sistema sólo recibirá un mensaje indicando que se ha encontrado un error y la acción no puede terminarse. Ante esta situación lo único que se puede hacer es parar la ejecución para volver a comenzar desde el estado inicial conocido.

Sin embargo, si junto con ese error se enviase también información acerca de su posición actual, esto podría resultar muy útil a la hora de replanificar. PELEA cuenta con un módulo llamado *LowToHigh*, encargado de recibir datos de bajo nivel y pasarlo a alto nivel. De esta forma, con las coordenadas en las que se ha detenido y mediante dicho módulo se introduciría un nuevo predicado que indicase el lugar en el que se encuentra el robot. A partir de este punto se tendría un estado actual a desde el que sería posible la replanificación.

6.4.6. Creación del dominio y problemas

Dejando atrás el dominio del *rover* utilizado para ejemplificar, se implementa un dominio en PDDL propio del problema, en el que se incluyen las acciones vistas en el apartado anterior. Dicho dominio será utilizado para la realización de pruebas.

Para modelar dicho dominio en PDDL se han creado los siguientes predicados:

Predicado	Descripción
(available ?x – robot)	El robot se encuentra disponible para su uso.
(enabled ?x – robot)	El robot tiene los motores encendidos.
(at ?x – robot ?y – waypoint)	El robot se encuentra en un punto determinado.
(connected ?x – waypoint ?y – waypoint)	Los puntos se encuentran conectados, es decir, el robot puede navegar sin dificultad de uno a otro.
(corner ?y – waypoint)	Dicho punto se trata de una esquina.
(move_forward_blocked ?y - waypoint)	En el punto determinado el robot no puede continuar avanzando hacia adelante.

Predicado	Descripción
(has_map ?x - robot)	El robot tiene el mapa creado.
(map_saved ?x -robot)	El robot ha finalizado la creación del mapa y lo ha guardado.
(error ?x - robot)	El robot se encuentra en estado de error.
(teleop_mode ?x – robot)	El robot está en modo teleoperación

Tabla 6.41. Descripción de los predicados

Con estos predicados se marcan las precondiciones y efectos a la hora de implementar las acciones vistas. Cada una de ellas ha sido modelada como se muestra en las tablas.

Acción	Activar motores
Parámetros	Robot
Precondiciones	El robot debe estar disponible y sin estado de error.
Efectos	El robot enciende los motores.

Tabla 6.42. Acción "activar motores"

Acción	Iniciar mapeo
Parámetros	Robot
Precondiciones	El robot debe estar encendido y sin estado de error.
Efectos	El robot tiene el mapa o parte de él creado.

Tabla 6.43. Acción "iniciar mapeo"

Acción	Finalizar mapeo
Parámetros	Robot
Precondiciones	- El robot tiene que tener el mapa o parte él creado. - No se debe haber detectado ningún error.
Efectos	Guarda el mapa generado.

Tabla 6.44. Acción "finalizar mapeo"

Acción	Navegar

Parámetros	Robot – Punto1 – Punto2
Precondiciones	- El robot debe tener el mapa guardado. - El punto 1 y el punto 2 deben estar conectados. - El robot debe estar en el punto 1. - Debe tener los motores encendidos. - Debe tener el camino despejado. - No debe estar en estado de error.
Efectos	El robot deja de estar en el punto 1 para situarse en el punto 2.

Tabla 6.45. Acción "navegar"

Acción	Girar
Parámetros	Robot – Punto
Precondiciones	- El robot tiene que estar situado en una esquina. - No puede avanzar hacia adelante. - No debe estar en estado de error.
Efectos	El robot sigue situado en la esquina, pero como ha girado ahora sí puede moverse hacia el frente.

Tabla 6.46. Acción "girar"

Acción	Modo teleoperación
Parámetros	Robot
Precondiciones	El robot se puede encontrar indistintamente disponible o en estado de error.
Efectos	El robot pasa a modo teleoperado.

Tabla 6.47. Acción "modo teleoperación"

Estas son las acciones y predicados que forman el dominio creado para vehículo P3-DX. La acción *avanzar* no ha sido modelada en el dominio debido a que es necesaria información de los sensores para saber los metros que ha avanzado en línea recta y, por tanto, el punto en el que se ha detenido.

Los problemas asociados a este dominio se forman mediante combinaciones de predicados que dan lugar a los estados iniciales, para los cuales habrá que especificar la meta a alcanzar.

6.4.7. Replanificación

Como se ha explicado en el apartado anterior, sin la información de los sensores resulta imposible replanificar de forma que se encuentre un plan alternativo que permita cumplir las metas de navegación establecidas. De esta forma, recibir un mensaje de error del robot deja al sistema en un estado impreciso.

En un primer momento se remedió este caso finalizando el sistema, de forma que si el servidor envía como respuesta que no ha sido capaz de completar una tarea, este se para en el punto donde se encuentre, dejando al robot inutilizado. Evidentemente esto es una situación poco deseable, por lo que se optó por una solución alternativa haciendo uso de la replanificación.

Para ello se ha establecido como requisito que el plan del robot siempre debe finalizar dejando al vehículo en modo teleoperación. Esto resulta útil, ya que una vez que el sistema retira su control, en lugar de dejarlo en punto muerto, da el control al usuario para que este pueda realizar sus acciones o guiarlo de vuelta.

De cara al sistema, introducir este requisito ha supuesto la solución en el caso de recibir un error desde el robot. Como en ese punto no se tiene información acerca de lo ocurrido, se realizan dos acciones:

- Eliminar el predicado *(enabled robot)*: una de las precondiciones de la acción *enabled_motors* es que el robot no se encuentre en estado de error para encender los motores. Puesto que se desconoce el tipo de error que ha surgido, se elimina esta condición del estado actual del problema .
- Añadir el predicado *(error robot)*: se añade información sobre el error en el robot.

Según la estructura del dominio, ante este nuevo estado se invalidarían la mayoría de acciones posibles, tales como navegar o girar. Esto obligaría a realizar la replanificación del problema para encontrar una solución alternativa.

En este punto la única acción posible resulta pasar al modo teleoperación, por lo que la solución del planificador será esta tarea. Suponiendo que el error haya sido que se ha encontrado algún obstáculo en el camino que le haya impedido llegar a su destino, se le enviará la tarea y pasará a modo teleoperación, dando el control al usuario para que lo dirija al lugar deseado y respondiendo en un nuevo mensaje que la acción ha sido cumplida. Entonces, el plan habrá sido completado.

Si por el contrario el robot ha sufrido otro tipo de fallo en su sistema que le impide pasar a modo teleoperación, la tarea será igualmente enviada, con la objeción de que en este caso no se recibirá mensaje de respuesta indicando que el vehículo ha podido pasar a este modo. En este caso el sistema terminará por *timeout* indicando que no ha recibido respuesta del servidor.

6.4.8. Carga en Raspberry Pi

Una vez el sistema ya se encuentra funcional, el último paso es exportarlo para introducirlo en la Raspberry Pi.

Durante el desarrollo se tuvieron tres configuraciones java para poder realizar la ejecución del sistema, donde cada una de ellas llamaba a la clase *main* pasándole como argumento uno de los módulos de PELEA: Monitorización, Ejecución o Soporte a la Decisión. Recordemos que el cliente se encuentra embebido dentro del módulo de monitorización, por lo que no es necesario llamarlo desde una nueva configuración.

Para realizar la exportación, entonces, se ha creado un *.jar* ejecutable por cada una de estas configuraciones, de forma que mantuviesen la misma funcionalidad.

A la hora de ejecutarlo habrá que indicarle a cada uno de ellos la ubicación de clase principal y el módulo que ejecutarán, de la siguiente forma:

java –cp <nombre_jar> <ubicación_main> <args>

Por ejemplo, el módulo de monitorización se iniciaría de la siguiente forma:

java –cp mainM.jar org.pelea.mainM –c ./configs/configurationDeclarative.xml –n M1

Los archivos necesarios para el funcionamiento, tales como los ficheros de configuración, el planificador y el dominio del problema se encuentran en carpetas adjuntas a la ubicación de estos archivos *.jar*, de forma que puedan ser accedidos cuando sea necesario.

Dichos ejecutables serán lanzados desde un *script* para que comiencen al unísono, facilitando así la ejecución del mismo. De esta forma será ejecutado el sistema en la Raspberry Pi.

Para su correcto funcionamiento en la placa se ha tenido que hacer una pequeña modificación en cuanto al desarrollo inicial, ya que se trata de una estructura distinta. La integración se ha llevado a cabo un ordenador Intel Core i7 de 64 bits, mientras que la placa Raspberry posee un ARMv7, de 32 bits. Esto supuso que el planificador utilizado por defecto en PELEA, *Metric-FF*, para su uso en 64 bits y previamente compilado devolviese un error al invocarlo desde el nuevo sistema. Por ello se introdujo en la placa una versión limpia del planificador en 32 bits y se compiló en el mismo sistema. Una vez generado el ejecutable, se cambió la llamada desde el sistema para que el módulo de Soporte a la Decisión iniciase dicha variante en lugar de la utilizada en un principio.

7. RESULTADOS

En este capítulo se muestran los resultados finales obtenidos mediante una serie de pruebas que comprobarán el funcionamiento del sistema de control desarrollado.

7.1. Pruebas unitarias

Estas pruebas se realizan sobre las unidades y funcionamientos más pequeños en los que se divide el sistema. Cubren los requisitos funcionales.

Cada una de las pruebas quedará especificada mediante la siguiente tabla:

Código	PU-XX	Requisitos	
Descripción			
Procedimiento			
Resultado esperado			
Resultado obtenido			

Tabla 7.1. Modelo tabla pruebas unitarias

- **Código**: donde PU son las siglas de Prueba Unitaria y XX el número de prueba en orden secuencial, en el rango [01,99].
- **Requisitos**: código del requisito o requisitos funcionales que son verificados con la prueba.
- **Descripción**: objetivo final de la prueba.
- **Procedimiento**: pasos realizados para la verificación.
- **Resultado esperado**: estado que se espera encontrar tras la realización.
- **Resultado obtenido**: resultado real alcanzado tras la prueba.

Código	PU-01	Requisitos	RF-01
Descripción	Comprobar que el sistema puede leer dominios en PDDL.		
Procedimiento	1. Introducir la ruta al dominio en PDDL 2. Introducir la ruta al problema en PDDL para observar que realiza acciones		
Resultado esperado	No se muestra ningún fallo o error.		
Resultado obtenido	Correcto.		

Tabla 7.2. PU-01

Código	PU-02	Requisitos	RF-02
Descripción	Comprobar que el sistema puede resolver problemas codificados en PDDL.		
Procedimiento	1. Realizar una ejecución normal y guardar el resultado (1ª comprobación) 2. Introducir un nuevo estado inicial 3. Resolver el problema.		
Resultado esperado	Al ejecutar el problema, el plan ha cambiado respecto al anterior, siendo capaz de resolver esta nueva variante.		
Resultado obtenido	Correcto.		

Tabla 7.3. PU-02

Código	PU-03	Requisitos	RF-03, RF-04
Descripción	El sistema lee los ficheros de configuración y en base a ello determina el estado a bajo nivel.		
Procedimiento	1. Establecer la ruta a los ficheros de configuración. 2. Comprobar que el estado ha cambiado a bajo nivel. 3. Verificar que a ese estado se le asocia el código de tarea.		
Resultado esperado	Se selecciona el estado a bajo nivel correspondiente al dado en alto nivel, y dicho estado es asociado con su código de tarea correspondiente.		
Resultado obtenido	Correcto.		

Tabla 7.4. PU-03

Código	PU-04	Requisitos	RF-05

Descripción	Si se introduce un error de formato en un fichero de configuración, el sistema avisará que se trata de un error humano en los datos de entrada.
Procedimiento	1. Introducir un error adrede en *customProblem.cfg* 2. Ejecutar el programa.
Resultado esperado	El sistema se detiene en el punto en el que accede al archivo, informando del error.
Resultado obtenido	Correcto.

Tabla 7.5. PU-04

Código	PU-05	Requisitos	RF-06
Descripción	El sistema continúa hasta el fin de la ejecución de forma automática.		
Procedimiento	1. Iniciar la ejecución. 2. Dejar al sistema actuar hasta la resolución del problema.		
Resultado esperado	El problema finaliza mostrando un resumen de los resultados.		
Resultado obtenido	Correcto.		

Tabla 7.6. PU-05

Código	PU-06	Requisitos	RF-07
Descripción	Si el cliente no logra contactar con el servidor se mostrará un error antes de iniciar.		
Procedimiento	1. Dar nombres erróneos no coincidentes al cliente y al servidor. 2. Iniciar la ejecución.		
Resultado esperado	Se muestra un error por consola.		
Resultado obtenido	Correcto.		

Tabla 7.7. PU-06

Código	PU-07	Requisitos	RF-08

Descripción	Se comprueba que las acciones son enviadas al cliente según el orden establecido en el plan.
Procedimiento	1. Iniciar el sistema. 2. Observar el plan creado por el módulo de decisión. 3. En los detalles en tiempo real del módulo de ejecución, comprobar el orden de las acciones.
Resultado esperado	El orden de las acciones gestionadas por el módulo de ejecución y enviadas a través del cliente es el mismo que en el plan mostrado por el módulo de Soporte a la Decisión.
Resultado obtenido	Correcto.

Tabla 7.8. PU-07

Código	PU-08	Requisitos	RF-09
Descripción	Comprobar que el sistema se mantiene en espera hasta que recibe respuesta por parte del servidor.		
Procedimiento	1. Marcar el momento del envío del mensaje 2. Esperar la respuesta del servidor.		
Resultado esperado	El sistema se suspende hasta que recibe contestación. En ese momento se reanuda el programa.		
Resultado obtenido	Correcto.		

Tabla 7.9. PU-08

Código	PU-09	Requisitos	RF-10, RF-11
Descripción	Comprobar que el sistema se detiene mostrando un error si la respuesta del servidor se demora más allá de un tiempo establecido.		
Procedimiento	1. Establecer el *timeout* en 7 segundos. 2. Enviar la respuesta del servidor a los 12 segundos.		
Resultado esperado	Pasados 7 segundos el sistema de detiene informando de que no ha recibido respuesta.		
Resultado obtenido	Correcto.		

Tabla 7.10. PU-09

Código	PU-10	Requisitos	RF-12
Descripción	Comprobar que el sistema recupera y reconoce la respuesta enviada por el servidor.		
Procedimiento	1. Recibir el mensaje de respuesta. 2. Acceder al contenido.		
Resultado esperado	El sistema pasa a gestionar la respuesta recibida.		
Resultado obtenido	Correcto.		

Tabla 7.11. PU-10

Código	PU-11	Requisitos	RF-13
Descripción	Comprobar que el sistema continúa ejecutando el plan en caso de respuesta positiva.		
Procedimiento	1. Recibir el mensaje de respuesta. 2. Reanudar la ejecución.		
Resultado esperado	Se envía al robot la siguiente tarea en el plan.		
Resultado obtenido	Correcto.		

Tabla 7.12. PU-11

Código	PU-12	Requisitos	RF-14
Descripción	Comprobar que el sistema replanifica si la respuesta recibida del servidor es negativa.		
Procedimiento	1. Recibir el mensaje de respuesta. 2. Reconocer que se trata de un error. 3. Añadir esta nueva condición al estado del problema.		
Resultado esperado	Se determina que no se puede continuar con el plan anterior y se obtienen las nuevas tareas posibles.		
Resultado obtenido	Correcto.		

Tabla 7.13. PU-12

Código	PU-13	Requisitos	RF-15
Descripción	Antes de finalizar el proceso, el sistema envía el mensaje necesario al robot para que este pase a modo teleoperación.		
Procedimiento	1. Dejar que el sistema resuelva el problema. 2. Comprobar que la última tarea enviada es “modo teleoperación”.		
Resultado esperado	Tras el envío de esta tarea el sistema finaliza.		
Resultado obtenido	Correcto.		

Tabla 7.14. PU-13

Código	PU-14	Requisitos	RF-16
Descripción	Comprobar que el sistema se detiene tras lograr sus metas y muestra un resumen del proceso.		
Procedimiento	1. Dejar que el sistema resuelva el problema. 2. Comprobar que los procesos se han parado.		
Resultado esperado	Todos los procesos se detienen, incluido el cliente. Se muestra un resumen de la ejecución.		
Resultado obtenido	Correcto.		

Tabla 7.15. PU-14

7.2. Pruebas de sistema

Una vez comprobado que los componentes funcionan de forma individual, en este punto se examina el sistema de forma conjunta, verificando así la integración de todos los componentes. Para ello se introducirán problemas modelados en PDDL y se plantearán una serie de casos para comprobar que la respuesta es la esperada.

El estado inicial simulará un mapa similar al mostrado en la imagen, en la que el objetivo será que el robot circule desde el punto 0, su situación inicial, hasta el punto 3.

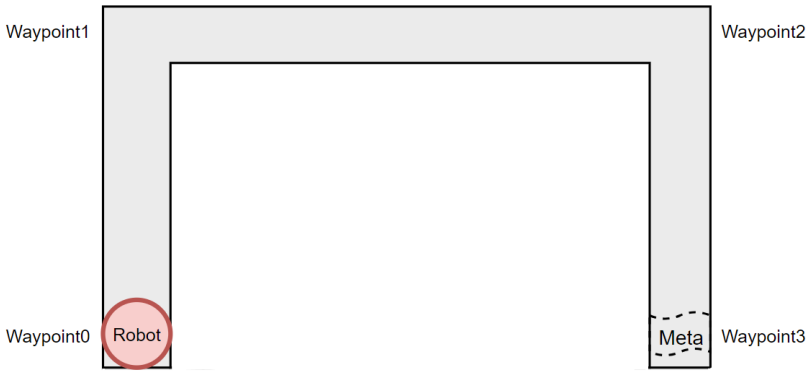


Fig. 7.1. Estructura del mapa de pruebas

Es una estructura sencilla que sin embargo resulta útil para comprobar que el funcionamiento del sistema es adecuado, ya que incluye, entre otras posibles tareas, navegación y giros. Crear un mapa más complejo simplemente alargaría la lista de acciones, pero no añadiría mayor dificultad.

Debido a que esta vez se incluyen imágenes, las pruebas no seguirán el mismo formato que las anteriores, pero se seguirá detallando el orden mediante un código *PS-XX*, donde *PS* es Prueba de Sistema y *XX* se trata del orden secuencial. De la misma forma, se especificará la descripción de la prueba, el resultado esperado y el resultado obtenido.

PS-01

Descripción: el robot ha realizado ya un mapeo de la zona, por lo que cuenta con conocimiento acerca de las coordenadas de cada punto.

Resultado esperado: el plan generado no incluirá tareas de mapeo, puesto que ya posee información al respecto.

Resultado obtenido: en la imagen de la izquierda, perteneciente al módulo de Soporte a la Decisión, aparece el plan calculado, cuya primera tarea tras encender los motores es navegar desde hasta el punto 2. Como se puede comprobar en la captura de la derecha, relativa al módulo de ejecución, dichas tareas se envían en orden al robot, sin que intervengan en ningún momento actividades de mapeo.

```
ff: search configuration is EHC, if that fails then  best-
    metric is  plan length

Cueing down from goal distance:    7 into depth [1]
                                   6            [1]
                                   5            [1]
                                   4            [1]
                                   3            [1]
                                   2            [1]
                                   1            [1]
                                   0

ff: found legal plan as follows

step    0: ENABLE_MOTORS ROBOT0
        1: NAVIGATE ROBOT0 WAYPOINT0 WAYPOINT1
        2: TWIST ROBOT0 WAYPOINT1
        3: NAVIGATE ROBOT0 WAYPOINT1 WAYPOINT2
        4: TWIST ROBOT0 WAYPOINT2
        5: NAVIGATE ROBOT0 WAYPOINT2 WAYPOINT3
        6: TELEOPERATION ROBOT0


time spent:    0.00 seconds instantiating 11 easy, 0 hard a
               0.00 seconds reachability analysis, yielding
               0.00 seconds creating final representation w
               0.00 seconds computing LNF
               0.00 seconds building connectivity graph
               0.00 seconds searching, evaluating 8 states,
               0.00 seconds total time

DEBUG [METRIC-FF]: Plan generated correctlly
```

Fig. 7.2. Soporte a la Decisión

```
>ENABLE_MOTORS ROBOT0
  >enable_motors

Low actions set: 0
 Task 0 response:
        Waiting for response...
 1 --> Ended successfully

Get sensor information to check the state of the world

>NAVIGATE ROBOT0 WAYPOINT0 WAYPOINT1
  >move_position waypoint1 degrees

Low actions set: 3 0 3 90
 Task 3 response:
        Waiting for response...
 1 --> Ended successfully

Get sensor information to check the state of the world

>TWIST ROBOT0 WAYPOINT1
  >twist_robot speed degrees direction

Low actions set: 2 10 90 180
 Task 2 response:
        Waiting for response...
 1 --> Ended successfully

Get sensor information to check the state of the world

>NAVIGATE ROBOT0 WAYPOINT1 WAYPOINT2
  >move_position waypoint2 degrees

Low actions set: 3 3 3 90
 Task 3 response:
        Waiting for response...
 1 --> Ended successfully

Get sensor information to check the state of the world

>TWIST ROBOT0 WAYPOINT2
  >twist_robot speed degrees direction

Low actions set: 2 10 90 180
 Task 2 response:
        Waiting for response...
```

Fig. 7.3. PS-01. Ejecución

PS-02

Descripción: el robot aún no ha reconocido la zona y no tiene ningún tipo de información.

Resultado esperado: antes de comenzar, será necesario realizar un mapa para contar con información acerca de las coordenadas.

Resultado obtenido: antes de comenzar a navegar el robot reconoce la zona y guarda el mapa.

```
ff: found legal plan as follows

step    0: ENABLE_MOTORS ROBOT0
        1: CREATE_MAP ROBOT0
        2: SAVE_MAP ROBOT0
        3: NAVIGATE ROBOT0 WAYPOINT0 WAYPOINT1
        4: TWIST ROBOT0 WAYPOINT1
        5: NAVIGATE ROBOT0 WAYPOINT1 WAYPOINT2
        6: TWIST ROBOT0 WAYPOINT2
        7: NAVIGATE ROBOT0 WAYPOINT2 WAYPOINT3
        8: TELEOPERATION ROBOT0


time spent:    0.00 seconds instantiating 11 easy, 0 hard
               0.00 seconds reachability analysis, yieldin
               0.00 seconds creating final representation
               0.00 seconds computing LNF
               0.00 seconds building connectivity graph
               0.00 seconds searching, evaluating 10 state
               0.00 seconds total time

DEBUG [METRIC-FF]: Plan generated correctlly
```

Fig. 7.4. PS-02. Soporte a la Decisión

```
Get sensor information to check the state of the world aft

>CREATE_MAP ROBOT0
  >init_mapping

Low actions set: 4
 Task 4 response:
        Waiting for response...
 1 --> Ended successfully

Get sensor information to check the state of the world aft

>SAVE_MAP ROBOT0
  >finish_mapping

Low actions set: 5
 Task 5 response:
        Waiting for response...
 1 --> Ended successfully

Get sensor information to check the state of the world aft

>NAVIGATE ROBOT0 WAYPOINT0 WAYPOINT1
  >move_position waypoint1 degrees

Low actions set: 3 0 3 90
 Task 3 response:
        Waiting for response...
 1 --> Ended successfully

Get sensor information to check the state of the world aft
```

Fig. 7.5. PS-02. Ejecución

PS-03

Descripción: Durante la ejecución de la tarea *twist_robot* en el punto 1 surge algún problema en el robot y este responde informando sobre ello.

Resultado esperado: Se evalúa el plan actual y se actúa en consecuencia, replanificando en caso de ser necesario.

Resultado obtenido: Como se ha añadido un error al estado actual, se descarta el plan seguido hasta el momento para dar el control del robot al usuario, estableciendo el modo teleoperación. En la imagen se puede ver un resumen del proceso. El módulo de Ejecución recibe el error tras la espera del mensaje, momento en el que Monitorización incluye los nuevos datos y modifica el plan, que es enviado a Soporte a la Decisión. Este módulo obvia el plan anterior y pasa a gestionar el nuevo estado encontrado, dando lugar a una única nueva tarea. Esta es enviada de nuevo al robot, que responde satisfactoriamente, resolviendo el problema y poniendo fin al proceso.

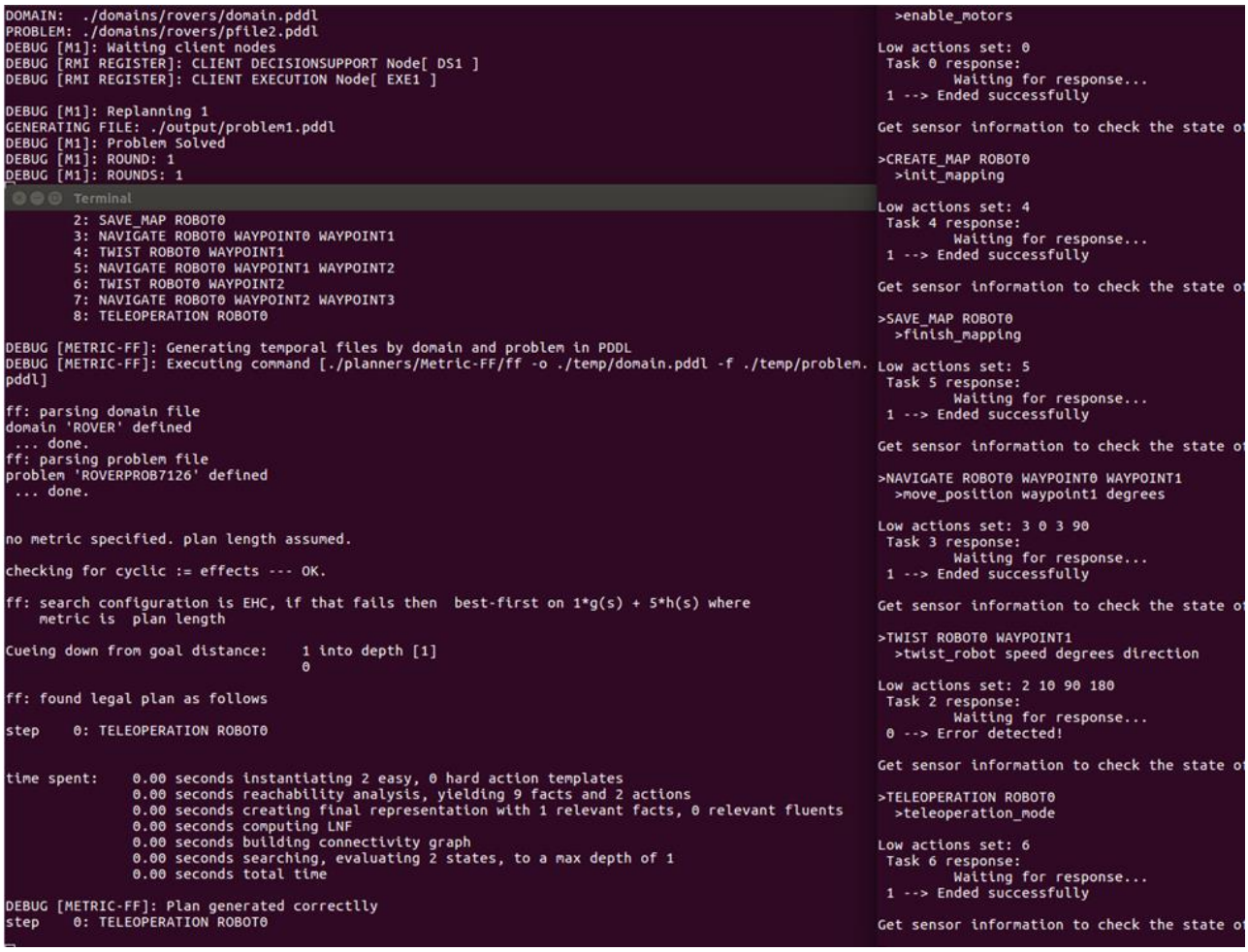


Fig. 7.6. PS-03. Monitorización, Soporte a la Decisión y Ejecución

PS-04

Descripción: En un caso similar al anterior, se detecta un error en una de las taras y es necesario replanificar. Esta vez el error se ha podido tratar de algún problema que impida al robot pasar al modo teleoperación.

Resultado esperado: El sistema espera suspendido hasta recibir la respuesta de la tarea. Como el robot no puede gestionar la respuesta se cumple el *timeout* establecido, finalizando el programa con un error.

Resultado obtenido: Tras la replanificación, se envía al robot la orden de pasar al modo teleoperación. Tras cumplir el tiempo límite de espera y no obtener ninguna respuesta, tal como se puede ver en la parte inferior derecha, se muestra el error detectado, indicando que se va a poner fin al servicio.

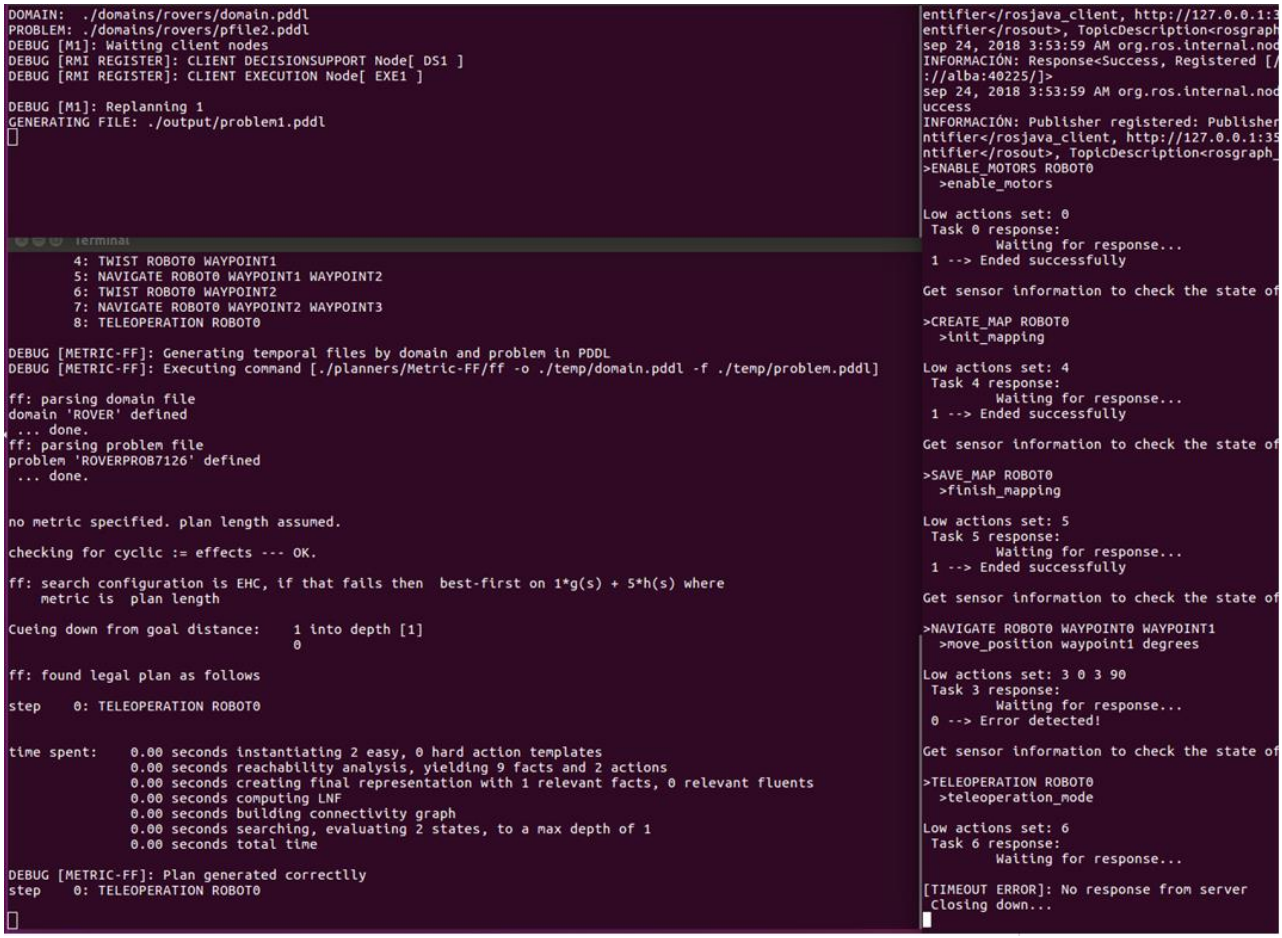


Fig. 7.7. PS-04. Monitorización, Soporte a la Decisión y Ejecución

7.3. Matriz de trazabilidad

Para comprobar que las pruebas de los módulos verifican la total funcionalidad del sistema se ha creado la siguiente matriz de trazabilidad, en la que se observa la relación entre las pruebas unitarias realizadas y los requisitos que cubre cada una de ellas.

	RF -01	RF -02	RF -03	RF -04	RF -05	RF -06	RF -07	RF -08	RF -09	RF -10	RF -11	RF -12	RF -13	RF -14	RF -15	RF -16
PU -01	X															
PU -02		X														
PU -03			X	X												
PU -04					X											
PU -05						X										
PU -06							X									
PU -07								X								
PU -08									X							
PU -09										X	X					
PU -10												X				
PU -11													X			
PU -12														X		
PU -13															X	
PU -14																X

Tabla 7.16. Matriz de trazabilidad pruebas unitarias - requisitos

8. CONCLUSIONES

8.1. Conclusiones generales

Tras la realización de este trabajo se ha conseguido integrar en una Raspberry Pi un sistema de planificación automática mediante el uso de PELEA y el paso de mensajes a través de ROS, para en proyectos posteriores ser utilizado para el control de un robot Pioneer 3 DX gracias a la interfaz de comunicación establecida entre ambos.

El sistema es capaz de gestionar el envío de mensajes de acuerdo a la planificación establecida, resolviendo así problemas planteados según las metas fijadas.

Hubiese sido interesante profundizar en algunos aspectos, pero debido a algunas limitaciones esto no ha sido posible. Un ejemplo de ello es la replanificación, ya explicado durante la interfaz de comunicación. Contar con los datos de los sensores hubiese supuesto poder modelar las excepciones de una forma mucho más completa y contemplando las distintas problemáticas que podrían plantearse. Pese a ello se ha logrado incluir la replanificación de tareas desde otra casuística, mencionando de forma teórica cuál sería el proceso en el resto de casos.

Otra limitación del trabajo ha sido el escaso dominio con el que cuenta para este problema el robot, ya que son pocas las acciones que puede realizar. Frente a otros dominios y problemas en los que se puede establecer como metas tomar fotografías, mover objetos, etc., que pueden dar lugar a planes más complejos, en este caso el vehículo tiene una serie limitada de tareas. Debido a esto se ha planteado un caso base que permita la verificación del funcionamiento del sistema, cuando también hubiese resultado curioso poder explotar más este ámbito.

Aún así, se ha conseguido el objetivo principal del proyecto, obteniendo como resultado un sistema funcional cuyo diseño ha sido pensado para ser fácilmente reutilizable para otros dominios o la ampliación de tareas.

8.2. Conclusiones técnicas

En cuanto a la forma de planteamiento del problema creo que ha sido correcta, tanto en las tecnologías utilizadas como en las alternativas que han quedado a libre elección, si bien es cierto que podrían haberse mejorado algunos aspectos que formarán ya parte de la sección de trabajos futuros.

Se ha demostrado que la unión de las dos herramientas principales utilizadas puede generar buenos resultados, y en manos de personas expertas multiplicaría su potencial.

Esto es en gran parte debido a PELEA, un sistema capaz de administrar dominios sin importar su contexto o dimensión. Por tanto, además de usarse dirigido al control de un robot P3-DX, como es el caso, este sistema podría ser implementado en proyectos más amplios. Con las nociones vistas durante el desarrollo de este trabajo es fácil predecir que su empleo dará buenos frutos. Pese a ello, durante la búsqueda de información se ha podido comprobar que se ha utilizado en escasos proyectos, todos ellos relacionados con el ámbito universitario y a modo de prueba de la herramienta, una situación que debería cambiar.

La parte negativa es quePELEA posee poca documentación a disposición del usuario, por lo que en un primer momento su uso e implementación puede volverse algo complicado.

En cuanto a ROS, se ha tratado de un acierto sin duda, ya que aunque se ha podido ver sólo una parte de la herramienta, cuenta con una gran comunidad en continua innovación, además de que su uso está implantado en grandes empresas. Frente al resto de alternativas de diseño comentadas posee más variedad de uso y facilidad de búsqueda de soluciones.

La Raspberry Pi resulta también una gran ventaja en cuanto a comodidad de uso, ya que permite la creación de un sistema portátil. Si hubiese que mencionar una desventaja sería su lentitud debido a su capacidad de RAM, que en ocasiones hacía algo tedioso su manejo, lo que hace pensar que con Arduino, que cuenta aún con menos capacidad de cómputo, hubiese resultado tarea imposible.

8.3. Conclusiones personales

Personalmente la realización de este trabajo ha servido para conocer de primera mano y de forma prácticas las posibilidades que ofrece la robótica y los múltiples mecanismos de implementación, un aspecto por el que había empezado a sentir curiosidad durante el desarrollo del Grado y gracias a asignaturas como Inteligencia Artificial y Aprendizaje Automático.

Conocía previamente ROS de un taller al que asistí durante el *T3chFest*, en el que pude aprender algunas nociones básicas sobre su uso. El desarrollo de este trabajo me ha permitido profundizar en estos conocimientos y emplearlos de forma práctica, realizando así un primer contacto con el mundo de la robótica.

En cuanto a PELEA, era un sistema sobre el que no conocía su existencia, por lo que en este punto se han producido algunas complicaciones.

La mayor dificultad encontrada ha sido el total desconocimiento de las herramientas utilizadas, ya que ha contado inicialmente con un proceso de estudio sobre ellas y

durante el desarrollo del trabajo continuas luchas a causa de no conocer al cien por cien el manejo de las mismas. Esto suponía prolongados estancamientos, lo que ha conllevado también retrasos en el desarrollo.

Por lo tanto, toda la fase de desarrollo ha supuesto un gran proceso de investigación, desde los inicios con ROS hasta la integración en la Raspberry Pi. Si bien es cierto que de no haber contado con las limitaciones comentadas se podría haber ahondado más en algunos aspectos, estoy satisfecha con los resultados ya que ha supuesto un gran aprendizaje en el cual he mejorados mis conocimientos teóricos y prácticos. Mediante el uso de las nuevas herramientas y métodos he sido consciente de la cantidad de combinaciones e ideas posibles de las que pueden surgir nuevos proyectos o innovaciones en los ya existentes.

Resumiendo, la realización de este trabajo me ha permitido ganar seguridad y competencias que me servirán en un futuro de cara a abordar nuevos proyectos.

9. TRABAJOS FUTUROS

Siguiendo lo dicho en las conclusiones, hay varios puntos que debido a falta de tiempo o limitaciones han sido vistos sin profundizar mucho de forma práctica en ellos, por lo que sería deseable poder seguir avanzando con ellos en posteriores trabajos.

De esta forma, una vez realizado el sistema de control, las líneas futuras de trabajo serían las siguientes:

- Conexión con el robot Pioneer 3 DX: en el momento en el que el sistema de control se encuentra funcional dentro de la Raspberry Pi, ya se encontraría listo para ser probado con el robot en un entorno real. Esto podría realizarse mediante una comunicación TCP/IP, en la que se conectaría la placa directamente al controlador el robot o a otra Raspberry que contenga el servidor. Como la interfaz ya está configurada para el paso de mensajes, no sería necesario ninguna modificación más aparte de configurar los parámetros de conexión.

- Información de los sensores: como se ha comentado en el apartado de desarrollo, una gran mejora consistiría en incluir información por parte de los sensores del robot. Esto conllevaría algo más de trabajo, ya que habría que modificar la interfaz de comunicación, pero a cambio se podría tener una herramienta con más potencial al permitir así la replanificación completa de tareas en caso de error, ya que se contaría con la ubicación en coordenadas del vehículo. Como ya se conoce el procedimiento teórico a seguir, incorporar esta mejora no conllevaría mucho tiempo.

- Añadir nuevos sensores y tareas: esto se trataría de un trabajo más costoso por parte del servidor, ya que incluir nuevos sensores supondría tener que gestionar más datos por su parte. En lo que respecta a este trabajo, bastaría con diseñar los predicados que modelen la nueva parte del dominio y añadir posteriormente las acciones. Una opción interesante sería conectar una cámara para la toma de fotografías. De esta forma se podrían tener planes más complejos que den lugar a una mayor experimentación.

10. PLANIFICACIÓN Y PRESUPUESTO

En este apartado se muestran las cuestiones relativas a la planificación del proyecto, mostrando las diferencias entre la esperada y final, junto con los costes totales del trabajo.

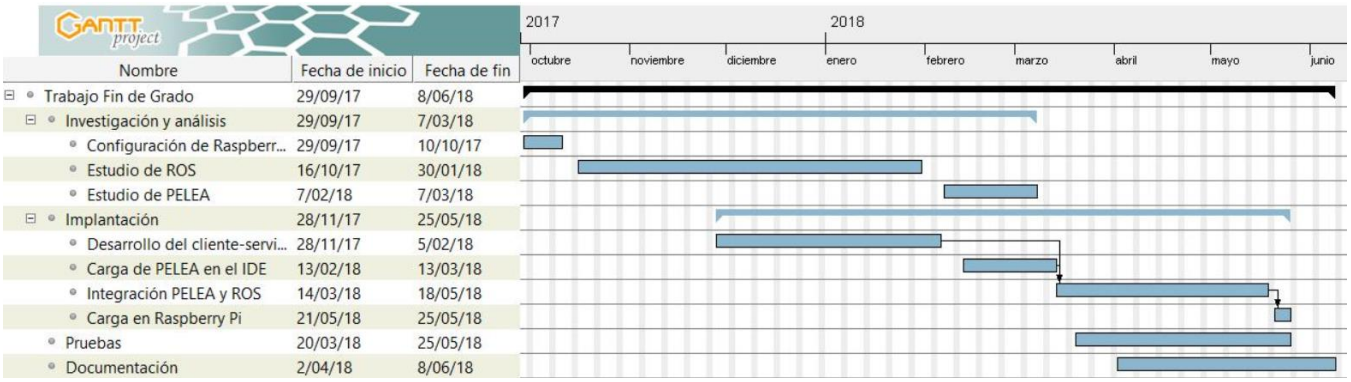
Las tareas incluidas en la planificación están basadas en el los objetivos y alcance del problema definidos en el apartado 3 de esta memoria. Por ello se pueden diferenciar cuatro bloques principales:

- **Investigación y análisis**: incluye la familiarización con las herramientas a utilizar, en un primer lugar mediante el estudio de la documentación y posteriormente mediante ejemplos prácticos seguidos a través de tutoriales en la mayoría de los casos. Durante dicho proceso se analizaron los sistemas y se tomaron algunas notas acerca de las posibles ideas de integración.
- **Desarrollo**: en este punto se comenzaron a realizar tareas de implantación del trabajo, dejando a un lado tutoriales y ejemplos. Dichas labores se han detallado en el punto 6.3.
- **Pruebas**: tiempo dedicado a comprobar que el sistema funcionaba según lo esperado. Aunque han existido pruebas finales este proceso se ha solapado con el de implantación, puesto que cada nueva función era probada mediante pruebas unitarias para detectar los errores de forma temprana.
- **Documentación**: aunque no se encuentra dentro de los objetivos principales forma parte de la planificación. Consiste en la escritura de esta memoria. Incluye también un proceso de investigación acerca del estado del arte, herramientas y demás datos y referencias que pueden leerse en este documento.

En base a estas tareas se han realizado los diagramas de Gantt que se pueden encontrar en los dos siguientes apartados.

10.1. Planificación inicial

Se trata de la planificación esperada cuando se comenzó el desarrollo de trabajo a finales de septiembre de 2017. Se trató de una planificación orientativa ya que se desconocía el trabajo real que iba a conllevar cada parte. Fue prevista para defender el trabajo en la sesión de julio 2018. En el diagrama Gantt se puede ver cómo se planificó el proyecto en un inicio.

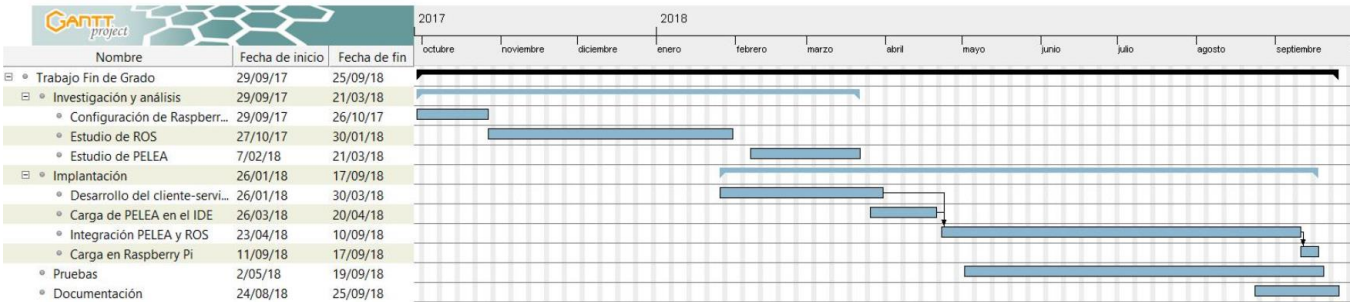


Nombre	Fecha de inicio	Fecha de fin
Trabajo Fin de Grado	29/09/17	8/06/18
Investigación y análisis	29/09/17	7/03/18
Configuración de Raspberr...	29/09/17	10/10/17
Estudio de ROS	16/10/17	30/01/18
Estudio de PELEA	7/02/18	7/03/18
Implantación	28/11/17	25/05/18
Desarrollo del cliente-servi...	28/11/17	5/02/18
Carga de PELEA en el IDE	13/02/18	13/03/18
Integración PELEA y ROS	14/03/18	18/05/18
Carga en Raspberry Pi	21/05/18	25/05/18
Pruebas	20/03/18	25/05/18
Documentación	2/04/18	8/06/18

Fig. 10.1. Gantt inicial

10.2. Planificación final

Tras la realización del proyecto, debido a variaciones esperadas respecto al tiempo invertido y a problemas surgidos con las herramientas y durante la integración, la planificación inicial sufrió algunas variaciones, por lo que se ha reestructurado el diagrama de Gantt para mostrar el tiempo real de cada tarea.



Nombre	Fecha de inicio	Fecha de fin
Trabajo Fin de Grado	29/09/17	25/09/18
Investigación y análisis	29/09/17	21/03/18
Configuración de Raspberr...	29/09/17	26/10/17
Estudio de ROS	27/10/17	30/01/18
Estudio de PELEA	7/02/18	21/03/18
Implantación	26/01/18	17/09/18
Desarrollo del cliente-servi...	26/01/18	30/03/18
Carga de PELEA en el IDE	26/03/18	20/04/18
Integración PELEA y ROS	23/04/18	10/09/18
Carga en Raspberry Pi	11/09/18	17/09/18
Pruebas	2/05/18	19/09/18
Documentación	24/08/18	25/09/18

Fig. 10.2. Gantt final

Como se puede ver, tiene una estructura similar a la planificación inicial, con la diferencia de que cada una de las tareas llevó más tiempo de lo esperado. El número de horas totales dedicado a cada fase puede verse en el siguiente apartado 10.3.

10.3. Costes

Para determinar los costes asociados a este trabajo se ha dividido el tiempo según los mismos bloques que en el apartado de planificación, de los que se han registrado el número de horas dedicadas a cada uno.

El coste por hora se ha calculado según un informe el elaborado por *InfoJobs* en el año 2017 sobre el estado del mercado laboral en España , en el que se habla sobre el incremento de vacantes en el campo de la robótica. Dependiendo del puesto, se requieren perfiles de formación profesional, ingeniero técnico, grado o ciclo formativo grado superior. El salario se sitúa en torno a los 29.000 euros anuales. Si al año hay unas 1.800 horas efectivas de trabajo, el coste por hora se sitúa entonces 16,11€.

Teniendo en cuenta estos datos se ha elaborado el coste del personal del proyecto, mostrado en la tabla.

Tarea	Nº de horas	Coste
Investigación y análisis	51	821,61 €
Desarrollo	192	3.093,12 €
Pruebas	11	177,21 €
Documentación	78	1.256,58 €
Total	**332**	**5.348,52 €**

Tabla 10.1. Número de horas del proyecto

El apartado de investigación y análisis correspondería al de formación de la persona que va a llevar a cabo el desarrollo. En un ámbito real, la persona encargada de implementar dicho sistema ya debería contar con los conocimientos necesarios para ello o necesitar menos formación, por lo que los costes podrían verse reducidos en dicho aspecto.

Respecto a los costes materiales, para el desarrollo e implementación de este sistema son necesarios los siguientes:

Material	Precio	Meses de uso	Meses en proyecto	Amortización en proyecto
Raspberry Pi Modelo B	35,00 €	3	3	35,00 €
Tarjeta Toshiba microSDHC UHS-I	12,00 €	3	3	12,00 €
RosJava	0 €	9	9	0 €
PELEA	0 €	6	6	0 €
Eclipse	0 €	72	9	0 €
Total	**47,00 €**			**47,00 €**

Tabla 10.2. Materiales

Todas las plataformas informáticas utilizadas se tratan de software de código abierto, por lo que no se ha tenido que pagar ninguna licencia por ello. De la misma forma, el desarrollo ha tenido lugar usando el Sistema Operativo *Linux*, por lo que tampoco ha conllevado ningún coste.

Por lo tanto, los costes finales del proyecto pueden resumirse en la siguiente tabla.

Medios	Coste
Humanos	5.348,52 €
Seguridad Social (33%)	1.765,01 €
Materiales	47,00 €
Subtotal	**7.160,53 €**
Coste indirecto (5%)	269,77 €
Total	**7.430, 30€**

Tabla 10.3. Coste total

11. Bibliografía

[1] D. G. Jerz, «R.U.R. (Rossum's Universal Robots),» Internet Archive, [En línea]. Available: https://web.archive.org/web/20060712035929/http://jerz.setonhill.edu:80/resources/RUR/.

[2] G. Cox, «A History of Robotics: Water Clocks,» Salvius Robot, [En línea]. Available: https://blog.salvius.org/2013/12/a-history-of-robotics-water-clocks.html.

[3] D. Mandal, «6 Automaton Conceptions From History,» Realm of History, [En línea]. Available: https://www.realmofhistory.com/2016/06/18/6-automaton-conceptions-history/.

[4] F. d. Bizancio, «Pneumatica,» de *Mechanikḗ Syntáxis*, Siglo III a. C..

[5] Just History Posts, «Magic and Robots: Medieval Automatons,» Just History Posts, 5 Abril 2017. [En línea]. Available: https://justhistoryposts.wordpress.com/2017/04/05/magic-and-robots-medieval-automatons/.

[6] J. A. Oliveira, «El Teleautomaton de Tesla: el primer dron marino,» Va de barcos, 18 Febrero 2017. [En línea]. Available: https://vadebarcos.net/2017/02/18/teleautomaton-tesla-primer-dron-marino/.

[7] Robot Hall Of Fame, «Robot Hall Of Fame,» Internet Archivo, [En línea]. Available: https://web.archive.org/web/20080706044437/http://www.robothalloffame.org/unimate.html.

[8] T. Trader, «Robotics Pioneer, George C. Devol, Dies at 99,» Enterprise Tech, 16 Agosto 2011. [En línea]. Available: https://www.enterprisetech.com/2011/08/16/robotics_pioneer_george_c-_devol_dies_at_99/.

[9] Robot Hall Of Fame, «Shakey,» Robot Hall Of Fame, [En línea]. Available: http://www.robothalloffame.org/inductees/04inductees/shakey.html.

[10] SRI International's, «Shakey,» Artificial Intelligence Center, [En línea]. Available: http://www.ai.sri.com/shakey/.

[11] InformáticaAplicadaAnaBlog, «Las 5 generaciones de la robótica,» 15 Enero 2016. [En línea]. Available: https://informaticaaplicadaanablog.wordpress.com/2016/01/15/la-5-generaciones-de-los-robotica/.

[12] J. C. Gómez y N. D. Muñoz, «Arquitectura hardware y software para un robot Rover,» *Avances en Sistemas Informáticos,* vol. 8, nº 3, pp. 183-189, 2011.

[13] National Aeronautics and Space Administration, «Mars Curiosity,» NASA, 4 Agosto 2017. [En línea]. Available: https://www.nasa.gov/mission_pages/msl/index.html.

[14] «Research Institute for Advanced Computer Science an Institute of the Universities Space Research Association,» [En línea]. Available: https://riacs.usra.edu/.

[15] «Universities Space Research Association,» [En línea]. Available: https://www.usra.edu/.

[16] «Ames Research Center,» NASA, [En línea]. Available: https://www.nasa.gov/ames.

[17] H. Utz, T. Fong y I. A.D. Nesnas, «Software Architecture for Planetary & Lunar Robotics,» [En línea]. Available: https://ti.arc.nasa.gov/m/pub-archive/1205h/1205%20(Utz).pdf.

[18] S. J. Russell y P. Norvig, Artificial Intelligence: A Modern Approach, 1995.

[19] M. Ghallab, D. Nau y P. Traverso, Automated Planning. Theory and practice, San Francisco: Elsevier, 2004.

[20] M. Stanek, «Historical intro to AI planning languages,» Machine Learnings, 26 Marzo 2017. [En línea]. Available: https://machinelearnings.co/historical-intro-to-ai-planning-languages-92ce9321b538.

[21] «International Conference on Automated Planning and Scheduling,» [En línea]. Available: http://icaps-conference.org/index.php/Main/HomePage .

[22] Arduino, «What is Arduino?,» [En línea]. Available: https://www.arduino.cc/en/Guide/Introduction .

[23] Arduino, «Arduino Products,» [En línea]. Available: https://www.arduino.cc/en/Main/Products.

[24] Raspberry Pi, «What is a Raspberry Pi?,» [En línea]. Available: https://www.raspberrypi.org/help/what-%20is-a-raspberry-pi/ .

[25] Raspberry Pi, «Downloads,» [En línea]. Available: https://www.raspberrypi.org/downloads/.

[26] Raspberry Pi, «Buy a Raspberry Pi,» [En línea]. Available: https://www.raspberrypi.org/products/.

[27] Á. G. Olaya, «Propuesta de integración de la arquitectura PELEA para el control de un robot,» 2013.

[28] Robot Operative System, «About ROS,» [En línea]. Available: http://www.ros.org/about-ros/.

[29] Robot Operative System, «Topics,» ROS, [En línea]. Available: http://wiki.ros.org/Topics.

[30] Robot Operative System, «Services,» ROS, [En línea]. Available: http://wiki.ros.org/Services.

[31] J. C. Mateu, «Desarrollo de un pequeño robot móvil basado en microcontrolador,» Septiembre 2014. [En línea]. Available:

https://riunet.upv.es/bitstream/handle/10251/43247/Memoria.pdf?sequence=1.

[32] Prometec, «Programación de sigue líneas,» [En línea]. Available: https://www.prometec.net/mblock-siguelineas-final/.

[33] Arduino Project Hub, «54 robot projects,» [En línea]. Available: https://create.arduino.cc/projecthub/projects/tags/robots .

[34] Aditya, «ROS robot,» hackster.io, 27 Febrero 2016. [En línea]. Available: https://www.hackster.io/adi1690/ros-robot-004bf8.

[35] «Poor Man's Raspberry Pi Turtlebot,» Hackaday.io, 1 Enero 2015. [En línea]. Available: https://hackaday.io/project/8629-poor-mans-raspberry-pi-turtlebot .

[36] S. Uddin, «Pioneer3AT Mobile Robot with UP Board and ROS,» Hackster.io, 13 Enero 2017. [En línea]. Available: https://www.hackster.io/salahuddin/pioneer3at-mobile-robot-with-up-board-and-ros-c1b43e.

[37] J. Hoffmann, «Metric-FF,» [En línea]. Available: https://fai.cs.uni-saarland.de/hoffmann/metric-ff.html.

[38] J. Hoffmann, «Fast-Forward,» [En línea]. Available: https://fai.cs.uni-saarland.de/hoffmann/ff.html.

[39] Robot Operative System, «RosJava Kinetic,» ROS, [En línea]. Available: http://wiki.ros.org/rosjava/Tutorials/kinetic .

[40] Robot Operative System, «RosJava Indigo,» ROS, [En línea]. Available: http://wiki.ros.org/rosjava/Tutorials/indigo/.

[41] Xataka, «Llega la nueva Raspberry Pi 3 Model B+,» Xataka, 14 Marzo 2018. [En línea]. Available: https://www.xataka.com/ordenadores/llega-la-nueva-raspberry-pi-3-model-b-mismo-precio-pero-mas-velocidad-y-wifi-de-doble-banda.

[42] MobileRobots, «Pioneer 3-DX,» [En línea]. Available: http://www.mobilerobots.com/Libraries/Downloads/Pioneer3DX-P3DX-RevA.sflb.ashx.

[43] Omron Adept Mobile Robots, «Pioneer 3 Operations Manual,» Abril 2017. [En línea]. Available: http://robots.mobilerobots.com/docs/all_docs/P3OpMan6_5.pdf .

[44] Eclipse Foundation, «About the Eclipse Foundation,» [En línea]. Available: https://www.eclipse.org/org/.

[45] M. K. Biradar, «Popularity of programming languages.,» Information SuperHighway, 28 Julio 2015. [En línea]. Available: https://maheshbiradar.wordpress.com/2015/07/28/popularity-of-programming-language/ .

[46] I. Burazin, «Most Popular Desktop IDEs & Code Editors in 2014,» Codeanywhere, 13 Enero 2015. [En línea]. Available: https://blog.codeanywhere.com/most-popular-ides-code-editors/.

[47] Mobile Robot Programming Toolkit, «MRTP Empowering C++ development in robotics,» [En línea]. Available: https://www.mrpt.org/.

[48] Microsoft, «Microsoft Robotics Developer Studio,» [En línea]. Available: https://www.microsoft.com/en-us/download/details.aspx?id=29081.

[49] RoboComp, «RoboComp. A simple robotics framework,» [En línea]. Available: https://robocomp.github.io/web/.

[50] Robot Operative System, "ROS answers," [Online]. Available: https://answers.ros.org/questions/.

[51] General Motors, «Cruise,» [En línea]. Available: https://getcruise.com/.

[52] International Business Machines, «IBM Watson Health,» IBM, [En línea]. Available: https://www.ibm.com/watson/health/.

[53] C. B. Frey y M. A. Osborne, «The future of employment: how susceptible are jobs to computerisation?,» 17 Septiembre 2013. [En línea]. Available: https://www.oxfordmartin.ox.ac.uk/downloads/academic/The_Future_of_Employment.pdf.

[54] H. Cohen, «Harold Cohen,» [En línea]. Available: http://www.aaronshome.com/aaron/index.html.

[55] CaixaBank Research, «¿Llegará la Cuarta Revolución Industrial a España?,» CaixaBank, 11 Febrero 2016. [En línea]. Available: http://www.caixabankresearch.com/llegara-la-cuarta-revolucion-industrial-a-espana-d3.

[56] CBS News, «The economic – and human – impact of the rise of robots and AI,» CBS News, 1 Mayo 2018. [En línea]. Available: https://www.cbsnews.com/news/the-economic-and-human-impact-of-the-rise-of-robots/ .

[57] Transparency Market Research, «Global Robotics Market: Manufacturers Vie for Product Commercialization to Attract Investments, observes TMR,» Octubre 2017. [En línea]. Available: https://www.transparencymarketresearch.com/pressrelease/robotics-market.htm.

[58] Parlamento Europeo, «Normas de Derecho civil sobre robótica,» 16 Febrero 2017. [En línea]. Available: http://www.europarl.europa.eu/sides/getDoc.do?pubRef=-//EP//NONSGML+TA+P8-TA-2017-0051+0+DOC+PDF+V0//ES.

12. ANEXOS

ANEXO A. RESUMEN EN INGLÉS

ABSTRACT

This report explains the development of a robot control system using Raspberry Pi and Automated Planning using ROS and PELEA. The first one is a framework for developing robotics software, which provides easier access by a set of libraries and structures that will be detailed in this document. The second one, PELEA, is a software created to integrate automated planning in robots that allows to monitor the execution and to know information about the state of the world.

First of all, the current situation and the historical area about tools and systems are known, such as robots, automated planning and Arduino and Raspberry Pi boards, technologies used in this project.

In this review, strengths and weaknesses of these technologies will be identified in order to guide the project implementation and to set the goals of it. In the socio-economic field, stands out the rise that robotics is having nowadays and the huge development that is still expected in the future, despite barriers to overcome. Later, the solution adopted to implement the system is described through a number of requirements.

Finally, the results and conclusions achieved after to finish the project are discussed, as well as information concerning to the planning and costs of the project.

Key words: Client; Server; Automated Planning; ROS; PELEA.

1. INTRODUCTION

For millennia, humans have been attracted by construction of machines that could reduce their workload or to be useful for any other activity, being the first automated clocks an example of this.

Through several designs and ideas, these concepts have advanced and have been developed over time to become we know as robotic. Especially during the last few years, this field is rising due to new developments and tools that compose it. Thus, we can be able to see how robotics went from being used to carry out repetitive tasks to begin to cover a wide range of works.

In this way came up Automated Planning, developed to control the *Shakey* robot, the first one which was able to move itself in autonomous way. Since then, Artificial Intelligence exists in people's daily lives in different ways, but this report is focus on robotics. Then, in these pages could be read about the "Fourth Industrial Revolution", that explains how it is impacting robot's insertion in the society, as well as the challenges to overcome.

The fact is that big progresses have been made thanks to this type of intelligent systems, such us the *Mars Science Laboratory* mission taken by NASA, which could place the *Curiosity* rover on the Mars's surface to take photographs and samples of the planet.

Noting this and other examples, it is clear that robotics and Artificial Intelligence fields are growing, often joining in projects that give rise to intelligent agents.

This is the origin of this project, whose goal is the develop of a system aimed at controlling a robot through Automated Planning. This document explains its development process, starting from the current and historical background of the technologies used, the socio-economic impact to finish with the results and conclusions obtained after completing the project.

2. STATE OF THE ART

2.1.Robotics

The word *robot* had its beginning in year 1921 in the woek R.*U.R (Rossum's Universal Robots)*, a theatrical performance written by Karel Čapek. This term comes from the Czech word *robota*, whose meaning is "forced work". After the success of the performance, this work began to be used in all languages, replacing the one used until then, *automaton*.

Centuries before Christ, were several inventors who tried to create their own automated designs. Since then, this field was progressing until year 1969, when Standford Research Institute built the first mobile robot which was able to make decisions, called *Shakey*. Equipped with sensors to engage with the environment and thanks to its system based on actions and goals, it was able to fulfil tasks such as navigating between points, turning lights on and off and manipulating objects.

From then to now, a large number of achievements have been made, linking with Artificial Intelligence to create bigger projects.

2.2. Automated Planning

Automated Planning is a branch of Artificial Intelligence which is able to build plans, that is, is able to generate a sequence of actions in such a way that them solve the problem. It is usually applied to control intelligent agents, like robots or unmanned vehicles.

In order to do this is required to provide some information through a formal definition. The most common is PDDL (*Planning Domain Definition Language)*, a predicate based language developed in 1998. Using this language characteristics of the problem can be specified, including status variables, actions and metrics, that allow to solve the problem under certain conditions, like consume the least amount of fuel possible in vehicle domain.

PDDL is a language based on actions, states and goals. The initial state and goals compose the problem, while actions (with preconditions and effects) and generic predicates build the domain.

2.3. Arduino and Raspberry Pi

As they say on their website: “Arduino is an open-source electronics platform based on easy-to-use hardware and software. Arduino boards are able to read inputs – light on a sensor, a finger on a button, or a Twitter message – and turn it into an output – activating a motor, turning on an LED, publishing something online. You can tell your board what to do by sending a set of instructions to the microcontroller on the board. To do so you use the Arduino programming language (based on wiring) and the Arduino software (IDE), based on processing.”

Used today for developing digital systems or interactive applications, it came up in 2003 as a tool aimed at students to take their first steps in electronic and programming.

As for Raspberry Pi, this is a small computer created by Raspberry Pi Foundation, engaged to educate young people and adults in the field of computer science and new technologies. Connected to a monitor and a keyboard, it is able to make same functions as any computer, as well as to interact with the environment and to connect several devices, being useful for a large number of projects.

Although they may look very similar, there are notable differences between them. While Arduino is a microcontroller, Raspberry is a small version of a complete computer. The first one board doesn’t support multi-tasking neither multi-threaded, so it is ideal for tasks that don’t need high computer power, such as some electronic project that involve LEDs.

The disadvantage of Raspberry Pi is that it needs an operating system to be able to use it, but on the other hand it has bigger computer power and memory, making it a good decision within complex projects.

3. AIMS AND SCOPE

The purpose of this work is the development of software control system for a P3-DX robot using a Raspberry Pi board and Automated Planning by means of ROS and PELEA.

Therefore, the system has been created in a reusable way, where every part is as independent as possible using tools that are easily implemented on different systems with similar hardware. For this reason, ROS is the ideal tool because of it allow to control the robot in an easy way through instructions, but in this case Automated Planning is going to be used, so that the vehicle could move automatically.

In this point comes up PELEA (*Planning Execution and Learning Architecture)*, a software developed for applying Automated Planned, by which different problems and domains can be resolve.

Then, the objectives are:

- Researching about ROS
- Researching about PELEA
- Implementation of ROS and PELEA
- Set up the domain and problems for P3-DX robot
- Configure the communication interface
- Load on Raspberry Pi board
- System testing
- Documentation

To sum up, with this project is created a robot control system by combining two different system, ROS and PELEA, in order to join the potential of both in the same tool.

4. MATERIAL AND METHODS

In this section tools used and the reason of its choice are explained, as well as alternative designs.

4.1.PELEA

PELEA, acronym of *Planning, Execution and LEarning Architecture*, is a tool created by some Spanish universities, including Carlos III University, to implement planning, control and automatic learning in robots, being able to monitor the execution of the plan.

This application consists of two level: the high level, composed of actions directly related with the planner, and the low level, linked to the tasks to be run in the robot.

PELEA consists of three main modules that run in parallel: Monitoring, Execution and Decision Support. These components share information to create the correct plan and manage necessary variables for high and low levels.

The general behaviour can be summarizing in the following scheme. This would be executed in a loop until the goals specified in the problem are met.

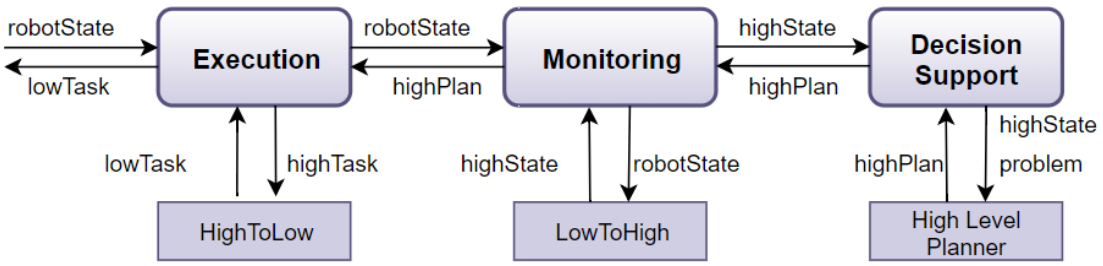


Fig. A. 12.1. PELEA operation

One of the advantages of this tool by which was chosen is that it is a system with large potential, and at the same time is independent of the domain and problem introduced, provided that the domain language be known by the planner. Is recommended to use PDDL, a standard language known by the most of planners.

4.2.RosJava

In its general version, ROS (*Robot Operative System)* is a framework for developing software to be implemented in robots. Thanks to a set of libraries and tools, it includes different tasks such as navigation, mapping, location, diagnosis and message passing.

The technique used within ROS has been a client-server structure, an easy and useful way to implement the communication between both systems. Server, in this case the robot, must be always available for any request from client. On its side, client, the developed system, has to send a request message with several information to be managed by the server. When server is finished, it must send a response message, including feedback about the task.

It runs in a loop until one of them stop the service, although the server can keep running without any client connected, but not the other way around.

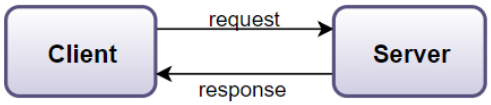


Fig. A. 12.2. Client-server scheme

For its implementation a ROS versión in Java has been used: RosJava, due to the fact PELEA is coded in the same language, as well as the previous known of it.

4.3.Raspberry Pi

After talk about the differences between Arduino and Raspberry Pi boards, for this work Raspberry Pi has been chosen, to make sure that there are no limitations during the project.

A Raspberry Pi 3 Model B of year 2016 has been used. In addition, to use a memory card SD has been necessary in order to save the Operating System and to load it on Raspberry Pi. The model is *Toshiba microSDHC UHS-I Card. 32 GB. Clase 10.*

4.4.Robot

It is a Pioneer 3 DX, a robotic base belonging to MobileRobots. In its simplest version it provides a front sonar, battery, wheels, a microcontroller and the Pioneer SDK software, which also provides a set of libraries and tools for the vehicle control, helping in the development process.

Also related with this project, Pionner 3 is a good option because of SDK package includes ARIA, a software for controlling data from robot. Thanks to this, the vehicle can be adapted for client-server structures, like in this case. In addition, ROS has a package called RosAria, compatible with the robot library ARIA, facilitate the integration process.

4.5.Eclipse

It is a programming IDE (*Integrated Development Environment)* widely used for programming in Java. It was developed by IBM until year 2004, when Eclipse Foundation was created, which is nowadays the administrator of the comminuty. His most well-known project is this IDE, which provides a wide range of tools such us a text editor and a compiler in real time, although these tools have also been extended in the same IDE for other languages like C or C++.

Due to the fact that its operation way was already known, its use was decided easily, saving in this way additional efforts. This is also a tool that provide full functionality to cover the project requirements, thanks to the compilation in real time and its debug system.

5. SOCIO-ECONOMIC BACKGROUND

Over the past few years, great advances have been made in the robotics world, becoming more and more visible in people's daily lives. Booming technologies such as Internet of Things, Automatic Learning, drones, 3D printers, assistance robots and so on are already making a new concept called "Fourth Industrial Revolution" or "Robotic Revolution".

In economic field, there are different theories and studies about it. When robotic when stablished in companies, there will be a decrease in cost and an increase in production, which will mean the product devaluation. The assumptions about this fact suggest that unemployment will be total, while others defend that only will only have jobs involved in technological area. This will produce a growing demand in these jobs, causing fall in wages.

Regarding this project, it is not expected to obtain any profit from it, it has just been developed as a final thesis to be implemented in a robot whose use is mainly for teaching purposes or research.

5.1. Regulatory framework

This project is under to Creative Commons Attribution – NonComercial – NoDerivatives license.

In addition, it is not under legal restrictions, because of at the moment there is no regulation that can be applied to robotics field.

It should be noted that due to the increase of this field, on 16 February 2017 a report with recommendations about civil law rules on robotics was approved by the European Parliament. It suggests different points on ethics, responsibility, security, registration of robots and another legal issues that will govern the future of robotics and artificial intelligence.

6. DESIGN AND IMPLEMENTATION

6.1. Use case

Once the system scope has been set and main components to use in the project are clear, different use cases that may be done during the use of system are detailed.

The use cases can be summarized in the following scheme.

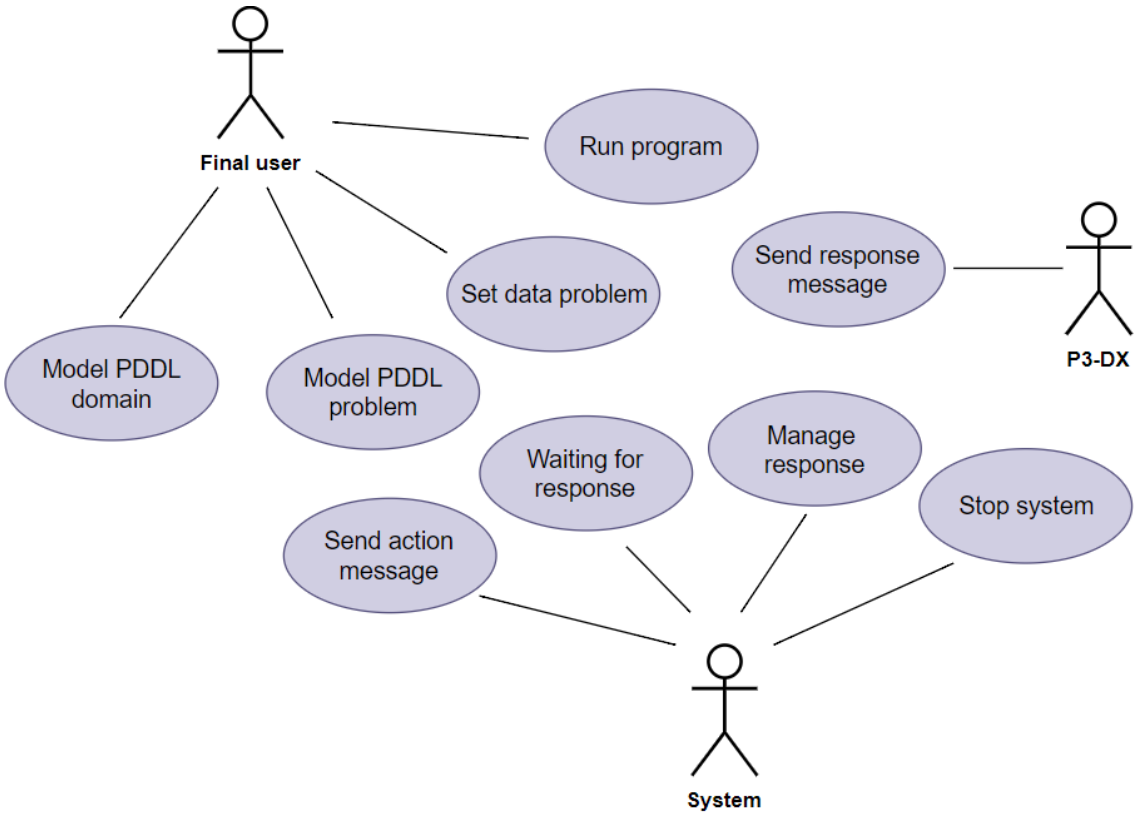


Fig. A. 12.3. Use cases

Where the following actors can be distinguished:

- End user: person who will use the system.
- P3-DX: Robot Pioneer 3 Dx.
- System: Developed module.

Based on these use cases, a list of requirements that the system must meet to verify its operation has been done.

6.2. Development

The following guides have been followed to develop the project:

- Load of PELEA: load of PELEA in the development IDE.
- Environment configuration: installation and configuration of RosJava for its use.

- Client-server structure: creation of the necessary structure for communicating the control system and the server. This structure has two messages: request and response messages. Both are aimed to send and return information between two sides.
- Communication interface: message creation with attributes that client and server have to exchange to execute tasks in the robot. With this structure and *genjava* the libraries that allow the communication with this interface are made. This interface will be the link between both systems.
- Domain and problem definition: create files in PDDL format in order to run the system. They include information that define the state of the world.
- Eclipse integration: join all elements in the development IDE to debug the project. In order to do this, new structures and elements to connect different modules have been necessary. During this time several tests to verify the correct behaviour of each module have been done. This allowed an early detection of problems, what facilitate its correction.
- Load on Raspberry Pi: export process and load the project on the board, to be more comfortable its link with the robot.

7. CONCLUSIONS

7.1. General conclusions

After completion of this work, an automated planning system using ROS and PELEA has been integrated into a Raspberry Pi, in order to communicate with Pioneer 3 DX robot in future projects thanks to the communication interface established between both.

The system is able to manage message sending according to the plan, solving in this way problems with certain goals.

It would be interesting to delve into more aspects, but due to some limitations this has not been possible. An example of this is the replanning. Having data from sensors would have meant to be able to model the exceptions in a much more complete way. Despite this, it has been possible to add replanning in another way, explaining theoretically what would be the process in the rest of cases.

Another limitation found has been the small domain that the robot has for this problem, because of it cannot made a wide range of tasks, compared with other domains in which different goals and tasks can be established, such us take photographs, move objects and so on. Due to this, a base case that allows verify the operation of the system has been proposed, but it would be also curious to be able to explore deeper this aspect.

In spite of this, the main objective of the project has been achieved, getting as a result a functional system whose design has been thought to be easily reusable for other domains or tasks extension.

7.2. Technical conclusions

Regarding the way of approaching the problem I think that it has been correct, both in technologies used and alternatives that have been freely chosen, although some aspects that will be part of the future work section could have been improved.

It has been demonstrated that the join of the two main tools can generate good results, and in expert hands would multiply their potential. On the one hand this is largely due to PELEA, a system aimed to manage different domains regardless of context or dimension. Then, in addition to being used to control a P3-DX robot, as in this case, this tool could be used in larger projects. With the basics seen during this work is easy to predict that its use will give good results. Despite this, it has been used in a few projects, all of them related to university environment and in a test way of the tool, a situation that should change.

On the other hand, PELEA has little documentation available to users, so at first its use and implementation can become some complicated.

About ROS, certainly its choose has been a success. Although we have seen just a part of the tool, it has a large community in continuous innovation, and it is used by several companies. Compared with the rest of the design alternatives discussed, it has a greater variety of use and ease of finding solutions.

Raspberry Pi is also a great advantage in terms of ease of use because it allows to make the system portable. If I would have to say a disadvantage it would be its slowness, due to its RAM capacity, which sometimes made its use tedious. This suggests that in case of using Arduino, which has less computing capacity, it would have been an impossible task.

7.3. Personal conclusions

Personally, to make this project has been useful to learn first-hand and in a practical way the possibilities offered by robotics and the multiple mechanisms of implementation, an aspect for which I begun to feel curious during my grade studies, thanks to subjects like Artificial Intelligence and Machine Learning.

I previously knew about ROS because I went to a talk during *T3chFest*, where I was able to learn some basic information about its use. The development of this work has

allowed me to deepen this knowledge and use it in a practical way, doing a first contact with robotics world.

About PELEA, it was a completely unknown system for me, so at this point there were some complications. The biggest problem found has been the total ignorance of the tools, because it has meant an initial study about them and a lot of trouble during the development, implying delays in the process.

The entire development phase has involved a big research process, from beginning with ROS to the integration into Raspberry Pi. Although if I had not had the limitations explained I had improved the project, I am satisfied with results because it has been a great learning in which I have improved my theoretical and practical knowledge. Using these new tools and methods I have been aware of the several possible combinations and ideas that can be implement.

To sum up, the completion of this work has allowed me to get security and skills that will result useful in the future in order to carry out new projects.

ANEXO B. MANUAL DE INSTALACIÓN

1. CONFIGURACIÓN DE LA RASPBERRY PI

a. Descarga del sistema operativo

El sistema operativo a instalar en la Raspberry Pi será Ubuntu Mate 16.04, debido que es una versión compatible con la última versión lanzada de ROS, Kinetic. Se puede descargar desde la siguiente ubicación: https://ubuntu-mate.org/raspberry-pi/

b. Carga en la tarjeta SD

Para instalar el SO en la placa, es necesario descomprimir la imagen e insertarla dentro de la tarjeta SD. Para ello se puede utilizar la herramienta Win32 Disk Imager.

c. Instalación de Ubuntu

Una vez grabada la imagen, basta introducir la tarjeta SD en la Raspberry y encenderla. Esta arrancará directamente con el menú de instalación de Ubuntu Mate. En este punto se trata de una instalación normal de Sistema Operativo.

d. Instalación de ROS

Para poder ejecutar posteriormente el sistema en la placa, será necesario que esta tenga instalado ROS, ya que para el envío de mensajes hará uso del *core* de la herramienta. No es necesario que se trate de la versión RosJava, puesto que ambas comparten las dependencias necesarias para que funcione su ejecución.

La versión a instalar será ROS Kinetic, compatible con el SO instalado. En primer lugar habrá que configurar los repositorios de Ubuntu para permitir *restricted*, *universe* y *multiverse*. Esto puede realizarse en la sección *Software & Updates* de Ubuntu.

Posteriormente es necesario configurar *sources.list* para aceptar la descarga de paquetes de ROS:

```
sudo sh -c 'echo "deb http://packages.ros.org/ros/ubuntu $(lsb_release
-sc) main" > /etc/apt/sources.list.d/ros-latest.list'
```

Y establecer las claves para el acceso:

```
sudo apt-key adv --keyserver hkp://ha.pool.sks-keyservers.net:80 --
recv-key 421C365BD9FF1F717815A3895523BAEEB01FA116
```

Una vez realizados estos pasos, ya se puede comenzar la descarga. Para las funcionalidades requeridas es suficiente la instalación de su versión de escritorio:

```
sudo apt-get install ros-kinetic-desktop
```

Para poder hacer uso de los comandos propios de ROS cada vez que se abra una nueva terminal, es conveniente hacer que las variables de entorno se añadan de forma automática:

```
echo "source /opt/ros/kinetic/setup.bash" >> ~/.bashrc

source ~/.bashrc
```

En este punto ya está configurada la Raspberry para poder ejecutar *roscore*, requisito para la correcta ejecución del sistema de control que será implantado posteriormente.

Se puede encontrar información ampliada acerca de la instalación de ROS en el siguiente enlace: http://wiki.ros.org/kinetic/Installation/Ubuntu

2. CONFIGURACIÓN DE ROSJAVA

a. Prerrequisitos

La versión de RosJava a instalar es Kinetic, resultando así compatible con la versión de ROS instalada en la Raspberry Pi, de forma que la aplicación se ejecute sobre una base correcta. Antes que nada es necesario instalar las dependencias necesarias:

```
> sudo apt-get install ros-kinetic-catkin ros-kinetic-rospack python-wstool openjdk-8-jdk
```

b. Instalación RosJava Kinetic

Para instalar al completo la versión, será necesario crear en primer lugar el directorio que contendrá todos los archivos y realizar en él la descarga:

```
> mkdir -p ~/rosjava/src

> wstool init -j4 ~/rosjava/src https://raw.githubusercontent.com/rosjava/rosjava/kinetic/rosjava.rosinstall
```

Mediante una serie de comandos se finaliza la ejecución:

```
> source /opt/ros/kinetic/setup.bash
> cd ~/rosjava
# Make sure we've got all rosdeps and msg packages.
> rosdep update
> rosdep install --from-paths src -i -y -rosdistro kinetic
> catkin_make
```

Para más información sobre la instalación de RosJava Kinetic se puede visitar el siguiente sitio web: http://wiki.ros.org/rosjava/Tutorials/kinetic/Source%20Installation

Si el uso de los comandos de ROS, tales como *roscore, rosrun…* presenta algún problema, ejecutar las siguientes líneas debería solucionar el error:

```
> sudo apt install ros-kinetic-roslaunch
> sudo apt install ros-kinetic-rosbash
```

c. Creación del cliente-servidor

Para la creación de la estructura a usar es recomendable seguir en un inicio el tutorial que se puede encontrar en la página oficial de ROS:
http://wiki.ros.org/rosjava_build_tools/Tutorials/indigo/UsingServices

d. Importación a Eclipse

Una vez seguido el tutorial y comprobado que el cliente-servidor funciona por terminal, se prepara para realizar su importación al IDE de desarrollo.

La forma más fácil de hacerlo es a través de *gradle*. Si no se dispone de la herramienta se pueden seguir las instrucciones de su página web:
https://docs.gradle.org/current/userguide/installation.html

Para que exportar el proyecto en un formato legible por Eclipse primero será necesario añadir este *plugin* al fichero *build.gradle* ubicado en /src/tutorial_custom_srv/client_server. Para ello basta con añadir la línea:

apply plugin: 'eclipse'

Luego, en ese mismo directorio, ejecutar:

```
> gradle eclipse
```

Cuando termine la exportación se podrá acceder a Eclipse y se elegirá la opción "Importar un proyecto ya existente", seleccionando desde la raíz el *workspace* que contiene la estructura cliente-servidor que se desea importar.

e. Ejecución en Eclipse

Con el proyecto ya importado a Eclipse, es hora de ejecutar el cliente-servidor para comprobar que se ha cargado correctamente. Si es así, ya se tendrán los archivos en un entorno de trabajo cómodo que permitirá su depuración.

Para ello se ejecutará como aplicación Java, eligiendo como clase principal: *org.ros.RosRun*. Como argumento habrá que pasarle la ruta correspondiente al nodo a ejecutar, en el caso del cliente: *org.ros.rosjava_tutorial_services.Client*

Como se trata de una aplicación ROS, en necesario tener ejecutando en una terminal el *core*, mediante el comando *roscore*. Adicionalmente habrá que añadir también las siguientes variables de entorno a Eclipse:

- ROS_ROOT
- ROS_PACKAGE_PATH
- PYTHONPATH
- PATH

f. Modificación del cliente-servidor

Una vez que se tiene la estructura funcional dentro de eclipse hay que modificar el código para adaptarlo al sistema a desarrollar. Si se ha seguido el tutorial, la función actual del cliente servidor corresponde al envío de un array de *x* posiciones. Es hora de desechar esa función para implementar el propio cliente, que haga uso de la interfaz de comunicación del robot.

En este punto PELEA ya debería estar importado también dentro de Eclipse, por lo que del mismo modo se podrían comenzar las labores de integración entre ROS y PELEA.

g. Creación de la interfaz de comunicación

Tal como se explica en la memoria del proyecto, la interfaz de comunicación debe tener la siguiente estructura:

```
int64 codigo_tarea
string[] vector_mover
string[] vector_girar
string[] vector_x_y
```

```
---
string feedback
int64  eval
```

Dejando por un momento a un lado Eclipse y volviendo al *workspace* de RosJava creado en el tutorial, para conseguir generar los mensajes que sigan este patrón se accede a /src/rosjava_custom_srv/srv. En su interior se crea un nuevo fichero de tipo *.srv*, llamado por ejemplo *ServerCustom.srv* que contenga la definición de la interfaz. Si el directorio aún contiene la definición del mensaje de ejemplo, borrar ese archivo.

Para generar el mensaje hacer uso de la herramienta *genjava*. Si no se tiene, se puede obtener mediante el comando:

```
> sudo apt-get install ros-kinetic-rosjava
```

Situado a nivel del directorio /src/rosjava_custom_srv/, introducir:

```
> genjava_message_artifacts –verbose –p rosjava_custom_srv
```

Esto generará un archivo *.jar* que contendrá la interfaz definida junto con los mensajes de petición y respuesta, creados ya con los atributos adecuados.

Para hacer uso de ellos sólo hace falta volver a Eclipse e importar dicho *.jar* como una librería nueva.

A partir de este punto la tarea consiste en finalizar la integración.